El libro de CODIGO MAQUINA del SPECTRUM



J. MARTINEZ VELARDE

J. MARTINEZ VELARDE

El libro de
CODIGO MAQUINA
del SPECTRUM

TERCERA EDICION

1986

PARANINFO S.A. MADRID

IMPRESO EN ESPAÑA
PRINTED IN SPAIN

ISBN: 84-283-1349-0
Depósito legal: M. 43.155-1985

PARANINFO S.A. Magallanes, 25- 28015 MADRID (5-3549)
Impreso en Lavel. Los Llanos, nave 6. Humanes (Madrid)

Indice

Parte primera

FUNDAMENTOS

Parte segunda

INSTRUCCIONES

Parte tercera

PROGRAMAS

APENDICES

DEDICATORIA:

A mi familia

Introducción

Las posibilidades que ofrece el ordenador doméstico ZX Spectrum (color, sonido, velocidad...) se aprovechan mínimamente en BASIC, lenguaje que tarde o temprano acaba por resultar lento y aburrido.

Sin embargo, el ZX Spectrum dispone de un misterioso y excitante lenguaje capaz de realizar, a la máxima velocidad, cualquier función que se le proponga. Estamos hablando del *código máquina*.

El usuario inteligente ha buscado por todos los métodos posibles informarse, superar el BASIC y acercarse al código máquina. Ha adquirido incluso libros ingleses por ser los únicos que hasta ahora existían sobre el tema, para introducirse en la programación de este lenguaje. Las consecuencias, por supuesto, no podían ser peores: no entendía el contenido por cuestiones idiomáticas, llegando a mezclar ideas y conceptos. En definitiva, el potente lenguaje pasaba a ser algo inescrutable y de suma dificultad. Como mucho se llegaba a saber sumar y restar unos números. Todos estos libros, que pretendían ser un glosario de las instrucciones del código máquina, carecían de ejemplos prácticos que pudieran ser estudiados y servir de prototipo para futuros programas.

Tienes en tus manos el verdadero libro de código máquina sobre el Spectrum. Se hace un estudio del lenguaje *desde un punto de vista práctico*, para que el usuario no sólo comprenda, sino que también *utilice* el código máquina.

El libro se divide en cuatro partes:

- La primera explica la base del lenguaje: los componentes, las formas de numeración, los primeros mnemónicos...

- La segunda introduce al usuario aún más en el lenguaje, mediante el estudio de una serie de instrucciones.

- La tercera parte profundiza en el código máquina práctico. Todo lo explicado anteriormente se utiliza en la presentación y explicación de rutinas y programas (entre los que hay que destacar el programa *Replicante y el Listado de Variables*).

  Estos programas servirán de base para los que el lector realice.

- La cuarta y última parte consta de una serie de Apéndices muy útiles para la programación.

El libro interesa al que desea abandonar el BASIC, buscando nuevos horizontes. El usuario debe incorporarse a la primera, segunda o tercera parte, según sean sus conocimientos previos (nulos, medios o amplios).

Sin duda, para todos los usuarios éste es el Libro de Código Máquina del Spectrum.

Juan Martínez Velarde
Madrid
Septiembre de 1984

Para la realización del original del libro se ha utilizado un ZX Spectrum 48K, un ZX Microdrive, un Interface ZX 1, un Interface Centronics, una impresora ADMATE DP-80, 10 Cartuchos de Microdrive de 90 K de capacidad media y el Procesador de Texto "Tasword Two".

Parte primera

FUNDAMENTOS

1

El "jefe" del SPECTRUM

Debajo del negro teclado y las minúsculas teclas, entre decenas de soldaduras, circuitos, elementos cuadrados con muchas patillas, se encuentra el "jefe del Spectrum".

Él es el que sabe todo acerca de la máquina: en que estado se encuentra, que es lo que tiene, o no que hacer, ...

Cada ordenador tiene un jefe de este tipo. Por supuesto, los ordenadores del mismo modelo tienen copias exactas del mismo sujeto. Estos cerebros de los ordenadores se denominan *micro-procesadores*, y existen varios modelos en el mercado.

El usado por el ZX Spectrum y su antecesor, el ZX81, y el antecesor de éste, el ZX80 . La "A" indica que es una versión más rápida del Z80.

El Z80 es un microprocesador de origen americano. Se construyó por la compañía Zilog, de California. Actualmente, el 60 de los ordenadores domésticos disponen de este microprocesador.

La energía que hace que el Z80 funcione proviene de la red eléctrica. El transformador externo modifica la tensión a 9V contínua. Dentro del ordenador se encuentra un segundo transformador, que varía la tensión, esta vez a 5V.

1. EL MICROPROCESADOR PUEDE HABLAR

1.1. Si-No

Se define como lenguaje la "capacidad de comunicación entre dos entes". Esta capacidad es la base del envío de información entre dos plantas (lenguaje vegetal), entre dos animales (lenguaje animal), entre dos personas (lenguaje humano), o entre una persona y una máquina (lenguaje de programación).

Cada uno de los seres que habitan en la tierra tiene un método para expresarse. y los ordenadores también.

El microprocesador no sabe nada de BASIC. Tampoco entendería el inglés, o la lengua usada entre nosotros, el español. Realmente el microprocesador se expresa en un lenguaje muy extraño, el *código máquina*. Precisamente este libro está dirigido a los que quieren aprender y utilizar este nuevo lenguaje.

No creas que el Z80 es un personaje algo alocado, por la forma en que habla. Si quiere decir un "Sí", envía una señal con tensión. Si quiere decir un "No", entonces no la envía. Lamentablemente no conoce otra forma de hablar!

Existe un método para "medir" la información (el número de síes y de noes). Tanto para medir alturas, pesos, información, es necesario definir la unidad básica, sobre la cual se crean las demás. No quiero meterme en temas que corresponderían de por sí a la Física, de manera que pasaremos directamente a definir la unidad de la información. Imagina una pregunta, que solamente tenga dos respuestas posibles, por ejemplo: "¿Luce hoy el sol?". Ante esta pregunta, esperaremos como respuesta un "Sí", o un "No". Para quien hace la pregunta, el "Sí" o el "No" tienen la misma probabilidad de salir; pues bien, la información que le transmite la respuesta "Sí" o la respuesta "No", es la unidad mínima de información.

Las unidades usadas para medir la cantidad de información se denominan *bits*. Su origen inglés viene de BInary digiTS (dígitos binarios) , pues precisamente en la base binaria existen dos, y sólo dos, cifras. Estas son el cero y el uno. La respuesta "Sí" puede estar representada por un uno. Este uso representará un bit positivo (igual a 1). La respuesta "No" puede estar representada por un cero. Este cero representará un bit nulo (igual a 0). Se dice en estos casos que el *estado* del bit es positivo (igual a 1) o negativo (igual a 0).

Si te has fijado, podemos interpretar el envío o presencia de tensión (que para el Z80 significaba un "Sí"), con un bit igual a uno. Por otra parte, el no envío o la ausencia de tensión (que para el Z80 significaba un "No") podrá ser interpretada con un bit igual a cero. De esta manera reemplazamos los "Síes" y los "Noes" del microprocesador, con unidades de información, con unos y ceros, respectivamente.

1.2. Pequeñas palabras

Piensa ahora en esta otra pregunta: "¿Qué día de la semana es hoy?". Existen, por lo menos, 7 posibles respuestas a esta pregunta. La cantidad de información que una de ellas aportará es mayor a un bit. El número exacto de bits es muy difícil de calcular, pero en cualquier caso, es bastante superior a uno.

Cualquier otra pregunta que no implique una respuesta "Si-No", espera en ella más de un bit de información.

El microprocesador Z80 tiene también la posibilidad de expresar más de un "Sí" o un "No", aportando más bits de información.

El número de bits de información que envía el Z80 para "hablar" con nosotros es de 8. Se ha elegido este número por una serie de razones que iremos conociendo poco a poco.

No puede enviarnos ni 5 ni 6 ni 7, tiene que ser exactamente 8 las unidades de información. Para este fin, el Z80 forma una serie de 8 bits, uno colocado junto al otro:

bit / bit / bit / bit / bit / bit / bit / bit

Para que no nos confundamos en el orden de los bits, cual es el primero y cual el último, éstos pueden ser numerados. El primer bit recibe el número 0, y de la serie de los ocho, es el que está situado más a la derecha. El número de orden de los bits se va incrementando a medida que tomamos en cuenta los bits de la izquierda. El último bit está numerado con el 7, y de la serie de los ocho, es el situado más a la izquierda:

bit7 / bit6 / bit5 / bit4 / bit3 / bit2 / bit1 / bit0

Este sistema, se numerar al primer elemento de un conjunto con el ordinal 0, es muy utilizado en este nuevo lenguaje.

Un bit puede adoptar la forma 0 ó 1, dependiendo del significado total del mensaje. Imaginemos los bits conteniendo ceros o unos, por ejemplo:

0 / 1 / 1 / 1 / 0 / 1 / 1 / 0

Esto equivaldría a:

Bit 0 = 0	Bit 4 = 1
Bit 1 = 1	Bit 5 = 1
Bit 2 = 1	Bit 6 = 1
Bit 3 = 0	Bit 7 = 0

Esta nueva forma de agrupar los bits desemboca en la formación de una nueva unidad para el Microprocesador. Esta nueva unidad es 8 veces mayor que el bit, y expresa mucha más información.

El nombre que recibe es un tanto confuso, por su similitud con la unidad mínima. Acabamos de conocer el *byte* (pronunciado <bait>), unidad con la que trabajaremos a partir de ahora.

El byte es mucho más útil, más exacto y más conciso que un simple bit. El byte permite que el Z80 se exprese a sus anchas.

En otros microprocesadores, por ejemplo el Z8000, también de la Zilog, los bits se agrupan en conjuntos de 16 bits. La información que un microprocesador de este tipo puede expresar en dos bytes (16 bits, también llamados "word") es muchísimo mayor. Los datos se manejan con más rapidez y comodidad.

El byte depende única y exclusivamente del *estado* (el valor) de cada uno de los 8 bits. Basándos en esto, cada byte diferente aporta una información determinada, y conocida por el microprocesador. Así pues, el byte con la forma 0 / 1 / 1 / 1 / 0 / 1 / 1 / 0

y en el lenguaje del Z80, el código máquina, significa una instrucción muy concreta. Cuando el ordenador recibe este byte, recibe una instrucción cuyo significado es: "Párate hasta que yo te avise!". El microprocesador se para, obedeciéndonos.

Existen otras combinaciones de unos y ceros en los bits. Por ejemplo, un byte con la forma

1 / 1 / 0 / 0 / 1 / 0 / 0 / 1

aportaría el Z80 otro mensaje. Su significado es el siguiente: "Abandona el modo de ejecución en código máquina y retorna al BASIC". Esta instrucción es muy útil, pues en el BASIC -lenguaje que se domina fácilmente- nos encontraremos más seguros que en código máquina.

Mediante este método de ceros y unos se expresa el Z80. Si aprendemos este lenguaje, nos veremos capacitados para "hablar" con él. Una serie de bytes, cada uno con un significado particular, formarían un programa. Este programa es comprendido al instante por el microprocesador, y nos obedece mucho más rápidamente que si se lo decimos en BASIC

¿Cuántos mensajes podemos aportar con un solo byte, que está formado por 7 bits? La respuesta es clara: exactamente el número de combinaciones de esas 8 cifras binarias. La primera sería la combinación 00000000 (8 ceros) y la última la 11111111 (8 unos). Si trabajáramos en base dos serían 100000000 (¡este número está en base 2!) combinaciones.

Supongo que te preguntarás cuántas combinaciones son 100000000 (en base dos) , en nuestra base decimal. El método para calcular es sencillo: se deben ir multiplicando cada cifra binaria por 2 elevado al n. de orden de 1, comenzando por la derecha y por el cero. De esta manera :

100000000 (2) =
= 1*2^8+0*2^7+0*2^6+0*2^5+0*2^4+0*2^3+0*2^2+0*2^l+0*2^0
= 256 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 = 256

Son exactamente 256 los mensajes, también llamados "palabras", que un byte puede adoptar . Ya conocemos dos, ¿Te acuerdas todavía qué forma presentaba el byte? Es verdaderamente difícil, si no imposible acordarse de ello, y sólo hemos conocido dos! No me imagino lo que puede ocurrir cuando queramos investigar más en este lenguaje. Necesitaríamos de una tabla, en la que buscaríamos el significado de cada byte, mirando su estructura binaria. Este sistema no es el más conveniente, y dificultaría mucho el aprendizaje del código máquina.

Una primera solución es interpretar el conjunto de bits, el byte, como una cifra binaria de 8 dígitos, y cambiarla de base dos a base decimal. Veamos lo que resultaría de esto:

El byte 0/1/1/1/0/1/1/0 se interpretaría como la cifra binaria 01110110, que se cambiaría de base:

01110110 = 0*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 0*2^3 + 1*2^2 +
1*2^1 + 0*2^0
= 0 + 64 + 32 + 16 + 0 + 4 +
2 + 0
= 118 decimal.

El byte 1/1/0/0/1/0/0/1 se interpretaría como la cifra binaria 11001001 , que al cambiarla de base:

11001001 = 1*2^7 + 1*2^6 + 0*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0
= 128 + 64 + 0 + 0 + 8 + 0 + 0 + 1
= 201 decimal.

Resulta muchísimo más sencillo acordarse que 118 decimal significa "¡ Párate hasta que yo te avise!", que 0/1/1/1/0/1/1/0 expresa el mismo mensaje.

Por otra parte, siempre se puede identificar el número 201 decimal con la orden "Abandona el modo de ejecución de código máquina y retorna al BASIC", lo que no se puede decir de 1/1/0/0/1/0/0/1.

A partir de ahora, y siempre que trabajemos en código máquina identificaremos y distinguiremos las cifras binarias de las decimales o de cualquier otra base por el siguiente sistema : los números decimales llevan tras sí la palabra "decimal", su abreviatura "dec", o solamente la letra "D".

Por otra parte, los números binarios van seguidos del término "binario", su abreviatura "bin", o la letra "B".

Una función incorporada al Spectrum, y con nombre "BIN", puede suplirnos el a veces aburrido trabajo de pasar los números binarios a decimales.

Introduciendo PRINT BIN 11001001, nos dará como respuesta 201.

Los 256 posibles mensajes o palabras diferentes, irían numerados del cero (correspondiente a 00000000 B, interpretación del byte 0/0/0/0/0/0/0/0) al 255 correspondiente al 11111111 8, interpretación del byte 1/1/1/1/1/1/1/1). El número de orden de estos mensajes se denomina *código*. A cada código corresponde una instrucción del microprocesador, y cada una de estas tiene su código correspondiente.

2. ¿ De qué dispone el microprocesador?

El Z80A del Spectrum no vive solo, no puede vivir solo. Necesita de una serie de cosas para cumplir la misión de obedecernos.

El microprocesador no sabe ninguna de las cuatro operaciones aritméticas básicas, por supuesto, un cálculo trigonométrico, ni cualquier función matemática que le programemos en BASIC o en código máquina. Por eso lleva siempre junto a él una pequeña calculadora, que le resuelve todo este tipo de problemas. Es posible, aunque complicado, acceder desde un programa en código máquina a la calculadora.

Es un microprocesador rápido, sobre todo si se lo decimos en su idioma, y eficiente, aunque no entiende una palabra de BASIC. Sin embargo, cuando se programa el ZX Spectrum en lenguaje BASIC, el ordenador ejecutará las órdenes que se le introduzcan, si no tienen fallos sintácticos.

Y es que el Z80 dispone de un diccionario. Cuando él lee la orden PRINT, va corriendo a buscar su diccionario. En el diccionario se le dice, en su idioma, dónde debe buscar para cumplirla y la ejecuta. Desgraciadamente, tras esto el Z80 tiene no conseguir acordarse nunca de nada de lo que acaba de hacer. De manera que tras obedecer al usuario, no recuerda para nada la orden PRINT , ni donde puede encontrarla. Si inmediatamente la vuelve a oir o leer, debe de ir de nuevo al diccionario para encontrar el comando, y allí se le dirá a dónde debe dirigirse.

El microprocesador trae consigo dos "libros". En el primero, el "gran libro de la sabiduría", están descritas, entre otras muchas cosas, todas las órdenes BASIC. El puede "leer" aquí qué es lo que debe hacer frente a un comando BASIC. El diccionario le indicará la "página" donde debe buscar. El Z80 se dirigirá directamente a dicha página, donde con pelos y señales y en su mismo idioma, el código máquina, estará escrito lo que tiene que hacer, Un conjunto de páginas indican al ordenador cómo debe "comportarse" en cada momento con el operador: qué colores debe usar, qué se le debe informar ...

Ni una letra de este libro puede ser modificada. Después de haber sido escrito, se le sometió a un preparado muy especial, de manera que ni siquiera la falta de energía eléctrica hará que todos estos datos se pierdan. Es casi, casi, un libro eterno.

Cada modelo de ordenador, aún con el mismo microprocesador, tiene su propio diccionario y su propio libro "imborrable". Esto es lo que caracteriza y diferencia un microordenador de otro: la calidad del diccionario y del libro (restringiéndonos a un aspecto interno).

El segundo "libro" es uno con la inmensa mayoría de sus páginas en blanco. En estas páginas podemos escribir nosotros lo que queremos decir al microprocesador, para que el lo lea. También podemos escribir el lenguaje BASIC.

En las primeras "páginas" del mismo, está escrito en una forma algo especial, como ya veremos, todo lo que aparece en la pantalla del televisor. Si no hay nada escrito en la pantalla, estas páginas permanecerán en blanco, vacías.

Una serie de páginas están guardadas para el microprocesador. Ya sabes que tiene una mala memoria, así que a veces, para que no se le olviden las cosas, escribe rápidamente de lo que se tiene que acordar y continúa con lo que estaba haciendo.

Este libro tiene la gran ventaja que la tinta se puede borrar sin ninguna dificultad. Podemos cambiar lo que queramos y cuando queramos. Otra ventaja es que si deseamos escribir algo en código máquina, podemos hacerlo donde queramos. Todas las páginas están a nuestra disposición.

La gran desventaja es que en el mismo momento en que no hay tensión (al desconectar), toda la información de este libro se pierde, y nos encontramos como al principio, vacíos.

Por esto mismo es necesario guardar en cinta magnética o en cartucho de Microdrive el contenido del libro. Equivaldría a "copiar" páginas y/o sus fracciones en una cinta.

Un dispositivo especial del ZX Spectrum hace que el microprocesador y sus "ayudantes" lean el teclado cada cierto tiempo. Esta operación se realiza 50 veces en un segundo.

Esto permite al usuario introducir sus órdenes, archivar sus datos, hablar con el Z80A ...

El microprocesador puede comunicarse con nosotros a través de la pantalla del televisor. Si está en ese momento trabajando en BASIC, en la parte inferior del televisor aparece un informe. Nos indica, por ejemplo, si falta memoria, si estás trabajando con números demasiado grandes, o si ha terminado con éxito un programa.

En cambio, si está ejecutando un programa en código máquina, solamente pueden aparecer dos tipos de informes. El primero es el 0, OK: todo ha terminado satisfactoriamente y el programa en código máquina es correcto. El segundo aparece cuando se detecta un error en el programa, o cuando los datos han hecho que el Z80 se confunda. Estos son los síntomas: la pantalla permancerá estática, ninguna tecla responderá... La única solución es desenchufar y volver a empezar.

Los dos libros sobre los que hablábamos antes va a ser ahora objeto de una mayor observación.

Ambos forman la *memoria* del microprocesador, pues él carece de ella. La memoria se define como aquel dispositivo que puede almacenar unidades de información, agrupadas en bytes, de tal foma que se pueda acceder y almacenar un conjunto de 8 bits.

Para tener una idea más clara, "acceder" tiene el significado de "leer" o "recuperar" una información determinada de un lugar de la memoria. "Almacenar" significa "escribir" o "introducir" información en la memoria. El número de bits que se leen o se escriben en, cada vez, de 8, es decir, un byte.

Cada uno de los "libros" representaba una parte de la memoria. El primera era la memoria denominada *ROM* (Read Only Memory, Memoria de Sólo Lectura). Los datos almacenados en la ROM sólo pueden ser leídos, es decir accedidos o recuperados. Solo se puede sacar bytes de la Memoria de Sólo Lectura. Esta es la memoria de más importancia de un ordenador, pues aquí está escrito cómo se deben ejecutar cada una de las órdenes BASIC ("SISTEMA OPERATIVO") y el diccionario ("TRADUCTOR BASIC").

La ROM del ZX Spectrum versión 16K es exactamente la misma que la ROM del ZX Spectrum versión 48K y ZX Spectrum + , por esta misma razón los programas para el 16K funcionan en un 48K y en un +.

Esta memoria está sellada, y nada puede afectar su contenido.

El otro libro es la memoria denominada *RAM* -externamente una letra, internamente un abismo de diferencia- (Random Access Memory, Memoria de Acceso Aleatorio, también llama-

da Memoria de Lectura/Escritura). El usuario o el microprocesador puede tanto escribir (introducir) como leer (recuperar) datos de esta memoria. Al escribir un programa, éste se almacena en RAM, y allí se mantiene hasta que se desconecte o lo borremos. El contenido de la RAM se pierde en el momento en que se desenchufa el ordenador. La estructura de esta memoria es muy extensa e interesante, y por eso se le ha dedicado el capítulo tercero.

La memoria, al igual que un libro, tiene una numeración determinada. Los libros comienzan por la página 1, y no tienen un final determinado: unos acaban en la página 100, otros en la 150, y otros en la 200.

La numeración de la memoria se realiza a través de *direcciones de memoria*. Cada una de estas puede ser interpretada como una página de un libro, que contendrá un sólo mensaje, una información determinada. Cada dirección de memoria tiene la facultad de contener un *byte*, unidad que equivale a 8 bits. El Z80 lee el contenido de una dirección de memoria, e interpreta como órden el byte, la información que contiene.

Para medir la memoria total, la memoria ocupada o la memoria libre de un ordenador, se usa una nueva unidad: el Kilobyte, por que el byte se nos queda pequeño. El kilobyte (K o Kb) no equivale a 1000 bytes, sino a 1024. La diferencia tiene una respuesta clara:

En nuestro sistema decimal, el Kilo, voz que tiene la significación de mil, es igual a 10 elevado a 3, porque 10 es la base que usamos en el sistema.

El Z80 usa el sistema binario, cuya unidad básica es el 2. 1000 nunca puede ser una potencia de dos, sino de 10. La potencia de dos más aproximada es 2 elevado a 10, que equivale a 1024.

Si se dice que un ordenador tiene 48K de memoria (48K = 48 * 1024 bytes = 49152 bytes) , lo que en realidad se quiere decir se puede contener 49152 bytes en su memoria, y que por ello tiene 49152 direcciones de memoria. El byte y el Kilobyte son unidades para medir *información*, no memoria. El origen de la confusión es que una dirección de memoria tiene espacio sólo para un byte, y el número de bytes que puede contener un ordenador es siempre igual al número de direcciones de memoria.

La primera dirección de memoria será siempre la numerada con el 0 dec. ¿Qué límite puede tener la memoria ?

Podríamos imaginar que las direcciones de memoria van aumentando: de la 0 se pasa a la 1 , de la 1 a la 2, y así sucesivamente ¿hasta cuándo?

Ciertamente hay un límite, que está mucho más cerca de lo que crees. El microprocesador dispone de sólo 2 bytes para almacenar una dirección de memoria. Esta es una de las características del Z80. Podemos calcular cuál sería la dirección más alta que el microprocesador puede controlar:

byte 1	byte 2
11111111	11111111

Esta equivaldría a 16 bits (8 de cada byte) con valor 1, numerados de 0 (derecha) al15 (izquierda). La dirección de memoria es entonces igual a 1111111111111111 bin (16 unos seguidos; la cifra resultante está expresada en base 2)" Si trasladamos este largo valor a base 10, lo entenderemos mejor.

$$
\begin{aligned}
1111111111111111 \text{ bin} = {} & 1*2^{15} + 1*2^{14} + 1*2^{13} + 1*2^{12} + 1*2^{11} \\
& + 1*2^{10} + 1*2^{9} + 1*2^{8} + 1*2^{7} + 1*2^{6} + 1*2^{5} + 1*2^{4} \\
& + 1*2^{3} + 1*2^{2} + 1*2^{1} + 1*2^{0} = \\
& = 32768 + 16384 + 8192 + 4096 + 2048 \\
& + 1024 + 512 + 256 + 128 + 64 + 32 + 16 \\
& + 8 + 4 \quad 2 + 1 = 65535 \text{ dec}
\end{aligned}
$$

La dirección máxima que puede alcanzar un ordenador con un Z80 es la 65535. En total dispondría de 65536 direcciones de memoria (la número cero también cuenta).

Estas 65536 direcciones contienen 65536 bytes (65536/1024 = 64K!). Las 65536 direcciones (¡como máximo!) son las que forman la memoria. La ROM dispone de una serie de estas direcciones de memoria, al igual que la RAM.

La memoria de sólo lectura (ROM) ocupa 16K (16K = 16 * 1024 = 16384 bytes) de los 65536 disponibles, un cuarto de la capacidad máxima. Esta memoria se encuentra ubicada a partir de la primera dirección (0 dec.) hasta la 16383, en total 16384 direcciones, que contienen el mismo número de bytes. La estructura de la ROM es la siguiente:

(1) Traductor BASIC	: aprox. 8 Kbytes.
(2) Sistema Operativo	: aprox. 7 bytes.
(3) Caracteres	: aprox. 1 Kbyte.

Los dos primeros conceptos ya fueron aclarados. El denominado "Caracteres" se refiere a una zona de memoria ROM, donde están definidos todos los caracteres de los que consta la máquina.

El conjunto de caracteres se llama Set o Juego de Caracteres. Para cada letra, número o símbolo se usan 8 bytes. Todos los caracteres están definidos al comenzar, porque sino no aparecerían ninguna letra, número o símbolo.

Existe un excelente libro, llamado "The Complete Spectrum ROM Disassembly" (El desensamblaje completo de la ROM del Spectrum), que es un análisis exhaustivo de esta memoria. Cada instrucción y grupo de instrucciones es aclarada. Resulta muy interesante al investigador avanzado del código máquina.

La memoria de lectura y escritura (RAM) es de diferente longitud en el ZX Spectrum modelo 16K y en el modelo 48K. Los dos se diferencias en la memoria de RAM. El primero ofrece 16K, 16384 direcciones de memoria. En este caso, ROM y RAM tendrían el mismo número de direcciones de memoria. El modelo ZX Spectrum+ ofrece igualmente 48K de memoria RAM.

El 48K y + ofrecen 3 veces más RAM. La capacidad de la memoria de lectura y escritura es de 48K, 49152 direcciones de memoria. La ROM es en los tres casos la misma (Un 16K se puede ampliar siempre a 48K, introduciendo la memoria de diferencia -32K-). La única diferencia entre uno y otro es su capacidad: en uno cabrán más instrucciones que en otro, pues tiene más memoria libre.

En los modelos 48K y +, ROM y RAM suman juntos 64K, 65536 direcciones de memoria. Por esta razón es técnicamente imposible ampliar un 48K o un + y disponer, por ejemplo de 80, para realizar programas de esa envergadura.

La RAM tiene marcado su comienzo siempre en la dirección 16384. Si hicieramos un esquema, situando las memorias por sus direcciones de memoria, veríamos que la RAM está "por encima" de la ROM. Con esto se quiere decir que las direcciones de memoria de la RAM serán siempre superiores a las de la ROM.

En el modelo de 16K, la RAM ocupará hasta la dirección (16384 + 16384) = 32768, ésta exclusive.

En el modelo de 48K y +, la RAM se expande hasta la dirección de memoria (16384 + 49152) = 65536, ésta también exclusive.

2

La base del Código Máquina

1. EL LENGUAJE ENSAMBLADOR

En el capítulo anterior decíamos que, dependiendo del estado de los bits, cada byte tenía un significado para el microprocesador.

Existen varios métodos para que los humanos podamos comprender el significado de los bytes. El primero que estudiamos interpretaba los bits como cifras binarias, que formaban números binarios en los bytes. Se debía cambiar el número de base 2 a base 10. Finalmente resultaban 256 posibles mensajes, numerados del 0 al 255. Estos reciben una denominación especial, los códigos.

Esta operación la realizamos con dos diferentes bytes, que daban lugar a dos códigos, el 118 dec. y el 201 dec. El primero comunicaba al Z80 que debía parar, hasta que se le avisara de nuevo. El segundo le indicaba el abandono del modo de ejecución en código máquina.

De todas maneras, si se tuviera que trabajar con números, difícilmente memorizables, pocas personas harían uso del código máquina. Esta es la razón de la creación de un nuevo lenguaje, que identifique con órdenes los diferentes mensajes del microprocesador. Estas órdenes tienen la función de facilitar la lectura de los mensajes. El conjunto de órdenes recibe el nombre de *mnemónicos*. Un mnemónico es cualquier método que "sirva para auxiliar a la memoria". En nuestro caso, el menmónico suple al número binario o al código decimal. La persona puede entonces identificar con mayor facilidad el tipo de mensaje.

El byte 0 / 1 / 1 / 1 / 0 / 1 / 1 / 0, cuyo código decimal era el 118 dec tenía la significación de "Párate hasta que yo te avise". El mnemónico que recibe se denomina HALT. Esta palabra inglesa tiene el significado de PARAR. El mensaje se identifica mucho más rápido por nosotros si lo vemos en la forma HALT, que en la forma 118 dec.

El otro byte 1 / 1 / 0 / 0 / 1 / 0 / 0 / 1 con el código decimal 201 dec y significación de "Abandono del modo de ejecución en código máquina" se le asigna el mnemónico llamado RET. También proviene de una palabra anglosajona RETURN, volver o abandonar.

Los mnemónicos suelen ser abreviaciones de sus mensajes correspondientes. En el Apéndice D del libro se encuentran tablas completas con todos los mnemónicos.

No existe mensaje sin mnemónico, ni mnemónico sin mensaje. Para cada mensaje existe un código decimal, y un número binario.

Mnemónicos, códigos o binarios son diferentes fórmulas para expresar lo mismo: un mensaje.

Realmente el código máquina, el lenguaje de un microprocesador, es el que trabaja con señales electrónicas binarias. La ausencia o presencia de tensión representan los estados lógicos 0 y 1. Este estado lógico contiene un bit de información. Ocho bits forman la unidad byte.

Todo aquello que suple al código máquina tiene la intención de facilitar la comprensión para las personas, pero hay que tener claro que un microprocesador no se expresa ni con códigos decimales, ni con mnemónicos.

El conjunto de mnemónicos y su uso forman parte de un nuevo lenguaje, el *lenguaje assembler o ensamblador*. Como en el resto de los lenguajes hay que guardar una determinada sintáxis. El assembler es la interpretación humana de los mensajes del ordenador. No es como el BASIC, muy fácil de entender y practicar. En el assembler se expresan los mensajes de tal manera, que a simple vista puedan ser captados por cualquiera.

El lenguaje ensamblador es el primer paso en la evolución de los lenguajes, a partir del código máquina. El BASIC o el COBOL están ya un par de kilómetros de distancias.

Este es el momento de conocer un tercer mnemónico. Se denomina NOP y tiene el código 00000000 b /0 d. El mnemónico es una abreviatura de No OPeration (Ninguna Operación).

El mensaje que expresa esta instrucción es de no hacer nada. Cuando el microprocesador recibe el código 0, se para momentáneamente, no hace nada más.

Cuando se enchufa el ordenador, gran parte de la memoria RAM contiene el código 0, correspondiente al mnemónico NOP. Cuando se quiere "borrar" código máquina, se suele utilizar este código, introduciéndolo en las direcciones de memoria que almacenan los bytes del programa.
¡Es una instrucción irremplazable!

2. NUMEROS DECIMALES

Conjuntamente con el lenguaje ensamblador se utiliza un método diferente para contar los números. La base utilizada no es la decimal, ni la binaria. Los números se cuentan en base 16. Este sistema aporta bastantes ventajas, sobre las que hablaremos más adelante.

La base hexadecimal (base 16) consta de 16 cifras. Las diez primeras son las mismas que las utilizadas en el sistema decimal: 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. Para las otras 5 cifras que faltan, se utilizan 5 símbolos: las letras A, B, C, D, E y F.

Esto nos lleva a que el número 10 decimal es el 0A hexadecimal, el 11 decimal es el 0B hexadecimal, ...

DECIMAL	HEXADECIMAL
10	0A
11	0B
12	0C
13	0D
14	0E
15	0F

¿Y como se dirá en hexadecimal el número 16 decimal?

El método es el mismo que utilizamos cuando se nos acaban las cifras en el sistema decimal. El número siguiente al 9 dec es el formado por la segunda cifra (el 1) en combinación con la primera cifra (el 0) . De esta manera, sabemos que tras el 9 dec viene el 10 dec. El siguiente a éste es el formado por la segunda cifra (el 1), en combinación con la segunda (el 1), el 11.

En base 16 es exactamente igual. Después del 0F Hexadecimal, viene el formado por la segunda cifra (el 1), en combinación con la primera (el 0). De esta manera, el 10 Hexadecimal es el que sigue al 0F Hexadecimal.

DECIMAL	HEXADECIMAL
16	10
17	11
18	12
...	...
25	19
26	1A
27	1B
28	1C
29	1D
30	1E
31	1F
32	20

Ante la posibilidad de confundir números decimales, hexadecimales y binarios (piensa que el 10 puede ser binario, decimal y hexadecimal, tres valores muy distintos) es norma común escribir tras un número binario la palabra "binario", su abreviatura "bin" o la letra "b". Después de un decimal aparece la palabra "decimal", su abreviatura "dec" o la letra "d". Lógicamente, tras un número hexadecimal debe ir la palabra "hexadecimal", su abreviatura "hex", o la letra "h".

Para pasar de base 16 a base 10 debes multiplicar cada cifra con una potencia de 16. La primera cifra será la de la derecha. El exponente inicial es 0, y va incrementándose a medida que se avanza a las cifras de la izquierda:

20H	=	0*16^0	+	2*16^1	=	0	+	32	=	32 d
8AH	=	10*16^0	+	8*16^1	=	10	+	128	=	138 d
D4H	=	4*16^0	+	13*16^1	=	4	+	208	=	212 d
FEH	=	14*16^0	+	15*16^1	=	14	+	240	=	254 d

El microprocesador puede contar, con un byte, números en el rango 0 a 255, pues sólo dispone de 8 bits. En el capítulo anterior vimos que con dos bytes (16 bits) se llegaba hasta el número 65536. Imagínate que al realizar un programa en lenguaje ensamblador deseemos escribir el número 153 dec. Si usamos la notación binaria, tendríamos que escribir 8 cifras. El número sería el 10011001. Si usamos la base decimal, el número de cifras que debemos escribir es menor, se usan 3 cifras.

El número 153 dec expresado en hexadecimal solo requiere de dos cifras. Sería el 99 hex. y de las tres posibilidades, la más "económica":

10011001 bin = 153 dec = 99 hex

La gran ventaja de la notación hexadecimal es que podemos expresar en esta base todos los números del rango 0 a 255 con solo 2 cifras. Compruébalo tu mismo con el número mayor, el 255 dec. Puedes utilizar los programas que acompañan el capítulo, pero sería más conveniente que practicaras el cambio de base "a mano".

La razón de esta enorme ventaja es que las combinaciones de 16 elementos (las 16 cifras) entre sí son exactamente 16^ 2 = 256, la misma cantidad de números que puede contener un byte. Este número 256 es también una potencia de 2,2^8 = 256.

Los números en base 16 (0/ h al FF h) pueden representar mensajes del Z80, al igual que lo haría un código decimal.

En este caso, el número hexadecimal recibe el nombre de *Código Objeto*.

Cuando en el capítulo pasado hablábamos de las direcciones de memoria que podía controlar la Z80, utilizamos para ello 16 cifras binarias, que representaban a 16 bits (2 bytes).

Al cambiar al sistema decimal, no hacía falta escribir tantos dígitos. En, esta base se usa, como máximo, 5 cifras, para expresar, por ejemplo, el número 65535.

Es aquí donde también se usa el sistema hexadecimal. El número de cifras que utilizamos para albergar números en el rango 0 a 65535 dec es sólo de 4. El número hex más grande que se puede expresar con 4 cifras es el FFFF h.

FFFF h = 15*16^0 + 15*16^1 + 15*16^2 + 15*16^3 =
= 15 + 240 + 3840 + 61440 =
= 65535 dec.

Las direcciones de memoria suelen expresarse en base 16. Cuando se usa una dirección de memoria inferior a la 00FF h / 255 d, de manera que realmente sólo se usa un byte, el de la derecha, es norma escribir el byte de la izquierda con contenido 00 h.

De esta manera puede identificarse más fácilmente con una dirección, y no con un dato.

Cuando se hace uso de dos bytes para expresar números en lenguaje ensamblador y no en código máquina, como veremos más adelante, los dos no tienen el mismo significado ni la misma importancia.

El byte situado a la izquierda es el más importante. Sus cifras se multiplicarán por 162 y 163, mientras que las del byte situado a la derecha se multiplicarán por 160 y 161.

El de la izquierda es el byte más significativo. Su contenido es más importante que el del byte de la derecha, el menos significativo.

Ejemplo: F00F h

F0 : Byte más significativo (Most Significant Byte - MSB)
0F : Byte menos significativo (Less Significant Byte - LSB)

3. NUMEROS NEGATIVOS

El microprocesador tiene la posibilidad de almacenar números negativos. Bajo determinadas circunstancias interpretará una serie de números como negativos y otros como positivos.

Para ello representa los números en *complemento a dos*, una forma diferente de codificar números, similar a la codificación binaria. Mediante este sistema se expresan números positivos y negativos en un determinado rango.

El elemento que diferencia unos de otros es el *estado del 7. bit*. Para hacernos una idea más clara, tendremos que expresar los números en base dos.

00010100 b	=	20 d	=	14 h
00110110 b	=	54 d	=	36 h
01011110 b	=	94 d	=	5E h
01111010 b	=	122 d	=	7A h
01111111 b	=	127 d	=	7F h

Todos los números hasta el 127 d representan números positivos, pues el estado de su bit 7 es igual a 0 . Lo cual quiere decir que la representación en complemento a dos para los números en este rango (0 a 127d) no difiere nada de su representación binaria.

Para expresar números negativos se siguen los siguientes criterios: El estado del bit 7 del número ha de ser igual a uno, esto define el signo negativo del mismo.

La expresión del número negativo puede ser realizada en diferentes sistemas. En el sistema complemento a dos se expresa tal y como aparecería en nuestro sistema. Así:

Número negativo	Complemento a dos
-1	-1
-10	-10
-128	-128

El problema aparece cuando queramos expresar este complemento a dos en decimal, hexadecimal o binario. Si queremos utilizar un complemento a dos en un programa, éste aparece en cualquiera de los sistemas mencionados anteriormente.

El método a seguir para expresar estos números en dec, hex o bin es el siguiente. Se debe restar el valor absoluto del número que queramos en negativo de 256d, 100h ó 100000000b, dependiendo de la base que se use. Veamos algunos ejemplos de números negativos decimales expresados en complemento a dos, y su correspondiente valor decimal y hexadecimal:

Número negativo	Compl.a 2	Dec	Hex
-1	-1	256-1=255	100-01=FF
-2	-2	256-2=254	100-02=FE
...	...	...	...
-34	-34	256-34=222	100-22=DE
-105	-105	256-105=151	100-69=97
...	...	...	...
-127	-127	256-127=129	100-7F=81
-128	-128	256-128=128	100-80=80

El número más bajo que se puede expresar será el -128, pues el valor binario correspondiente a su complemento a dos es 10000000 b. Si quisieramos expresar el número -129, obtendríamos el valor binario 011111111, cuyo séptimo bit es igual a 0, y es requisito indispensable que sea igual a 1.

En el sistema de complemento a dos se interpreta el 255d como un -1, el 254d como un -2, y así hasta el 128d, que será interpretado como un -128. Los números se mueven por tanto en el margen -128 a + 127.

El apéndice C del libro contiene una tabla con las equivalencias decimal-hexadecimal-complemento a dos.

Observa el resultado de sumar el valor absoluto de un número negativo con el valor decimal correspondiente a su complemento a dos:

/-1/	+	255	=	1	+	255	=	0
/-2/	+	254	=	2	+	254	=	0

El resultado es igual a 0. Por ello se dice que ambos números se "complementan uno al otro". En realidad el resultado es igual a 256, pero para expresar este número (100000000 b) necesitaríamos 9 bits, y sólo disponemos de 8. El valor de esos 8 primeros bits es igual a 0. Esta es la razón por la que 1 + 255=0.

Existen otros sistemas para explicar la interpretación de números negativos con el complemento a dos, pero sin duda éste es el más sencillo.

La gran duda que seguramente tienes ahora es la siguiente:

¿Cómo sabe el microprocesador que, por ejemplo FFH, es igual a -1, y no a 255d o viceversa, si ambos tienen la misma forma?

La razón es que el Z80 tiene una serie de instrucciones que usan números positivos "normales" en el rango 0 a 255, y una serie de instrucciones que utilizan números en complemento a dos, en el rango -128 a + 127. La interpretación que se le de a un número depende por tanto de la instrucción que opere con él.

4. PROGRAMAS EN CODIGO MAQUINA. COMO ACCEDER A ELLOS

Un programa en código máquina se define como una serie de instrucciones o sentencias, que dispuestas de forma adecuada, definen una operación que la máquina debe realizar.

El programa debe tener como propósito alcanzar un objeto determinado. Esta definición no implica cual deba ser ese ese resultado. La función de un programa es materia y competencia del programador.

Una instrucción de microprocesador se define como un conjunto de caracteres que definen una operación. Las instrucciones pueden expresarse por medio de un número binario, un número decimal, un número hexadecimal, un mnemónico o una expresión, como ya hemos visto. En el lenguaje assembler se usan los mnemónicos y los números hexadecimales. Los mnemónicos HALT, RET y NOP son instrucciones.

Una operación es la acción que realiza la máquina para cumplir una instrucción. La operación correspondiente a la instrucción HALT es la parada del microprocesador, hasta que se le avise de nuevo.

Las instrucciones deben introducirse en la memoria del ordenador en forma de códigos. A cada instrucción corresponde un código determinado. La instrucción RET tiene el código C9 h o 201 d.

Existen otro tipo de instrucciones que ocupan 2,3 ó hasta 4 direcciones consecutivas de memoria. Estas instrucciones constan de 2, 3 ó 4 códigos. Estos formarán *un solo mnemónico*.

Al ejecutar el Z80 el programa, lee los códigos de las instrucciones, interpreta los bits que forman ese código, y realiza las tareas necesarias para completar la operación definida por la instrucción.

Existen dos comandos BASIC especiales para este fin: PEEK y POKE. Son dos órdenes opuestas. La orden POKE introduce en una dirección de memoria un valor determinado. Este valor puede ser el de un código, correspondiente a una instrucción. Tiene este formato: POKE dirección, valor. El primer argumento define la posición dentro de la memoria, la dirección donde se debe cargar el segundo argumento.

Imagina que queremos introducir en la dirección de memoria 7000 h / 28672 d el valor C9 h/201 d. Nosotros sabemos que corresponde a la instrucción RET. Para ello deberíamos escribir:

POKE 28672,201

La orden POKE del Spectrum sólo admite los parámetros en base decimal. Tras dar RUN, la dirección de memoria 28672 contiene el valor 201.

La función PEEK, tiene la función de averiguar el contenido decimal de una dirección de memoria, sin modificarlo. Si introducimos:

PRINT PEEK 28672

nos responderá 201, porque este valor fue el que introdujimos anteriormente.

Una vez esté hecho un programa en lenguaje ensamblador deben encontrarse los códigos decimales correspondientes a cada instrucción, e introducirlos a partir de una dirección de memoria. Otra solución es utilizar un programa ensamblador. Este tiene la ventaja, que nos trasnsforma directamente los mnemónicos en códigos, y los introduce en la memoria.

De las dos memorias que dispone el Z80, la ROM y la RAM, solamente está a nuestra disposición de programar la segunda. En la memoria de solo lectura no podemos introducir ningún dato, porque no se puede modificar el contenido de ninguna de sus direcciones de memoria. La ROM ocupa las direcciones 0000 h / 0 d a 4000 h / 16384 d. Probemos a introducir aquí un dato:

POKE 1, 201

Para comprobar si esta orden ha funcionado, debemos usar la función PEEK:

PRINT PEEK 1

Pero da como resultado 175, y no 201. Intenta hacerlo con otras direcciones de memoria inferiores a 16384. No lo conseguirás ...

Para introducir códigos están a nuestra disposición la memoria RAM. El ejemplo que nos sirvió para introducir PEEK y POKE usaba una dirección de la RAM. Prueba a introducir otros valores, y verás cómo aquí si funciona.

Todavía no hemos explorado la RAM, pero si quieres experimentar con otros valores y otras direcciones de memoria, borra cualquier programa que tenga ahora en la memoria, y usa direcciones superiores a la 24000.

Una vez se encuentren las instrucciones de un programa en código máquina en la memoria del ordenador, se puede acceder a él.

Existe una serie de funciones y comandos BASIC especiales para este fin, es decir existen diferentes métodos para poner en marcha un programa en código máquina.

La función común a todos estos programas es *USR*. Esta tiene la significación de USeR (Usuario). Tras esta función, el usuario debe especificar una dirección de memoria, en base decimal.

El microprocesador abandonará entonces el modo de ejecución BASIC, y pasará a interpretar como órdenes los códigos que se encuentren a partir de la dirección especificada.

Al abandonar el modo de ejecución de código máquina (al procesar una instrucción RET), se retorna con un argumento numérico. Este número depende de una serie de factores que serán discutidos en el capítulo cuarto.

A continuación, las formas más usuales de acceder a un programa en código máquina :

(1) RANDOMIZE USR dirección. Se procesa un programa en código máquina, ubicado a partir de la dirección especificada. Imagina que el número con el que se retorna al BASIC es el 255 d. Después de esto se procesaría el comando BASIC :

RANDOMIZE 255

pues la función USR dirección tuvo como resultado 255 d.

El comando RANDOMIZE afecta directamente a una variable del sistema (FRAMES) que origina la función RND. Esta es pseudo-aleatoria, pues depende del contenido de FRAMES, otra variable del sistema.

Puedes comprobarlo desenchufando y volviendo a enchufar el aparato. Al principio FRAMES tiene un valor determinado, y el primer RND será siempre el mismo, exactamente el valor 0.0011291504.

(2) PRINT USR dirección. Esta orden pone también en marcha un programa en código máquina a partir de la dirección que se especifique. El valor que trae consigo la función USR es impreso en pantalla, dada la orden PRINT.

(3) LET L = USR dirección. El microprocesador ejecuta un programa en código máquina a partir de la dirección nombrada. Una vez se halla retornado al BASIC, se inicializa la variable L con el valor que USR ha traído consigo.

De entre estas 3 posibilidades, la mejor es la primera, pues no nos modifica el contenido de la pantalla, ni utiliza una variable BASIC. Sin embargo, las opciones 2 y 3 son a veces las escogidas, precisamente por sus consecuencias (para mostrar o guardar en una variable ese valor de retorno).

La dirección que se especifique debe estar entre 0 y 65535.

Con todo esto ya podemos realizar nuestro primer y pequeño programa en código máquina. Constará de una sola instrucción: RET . Comprueba primero que el ZX Spectrum no contiene nada en su memoria (haz un NEW) y. sigue leyendo.

La dirección de memoria escogida es la 7000 h / 28672 d. Esta es una zona de la memoria RAM totalmente libre y segura.

El código perteneciente al mnemónico RET es el C9 h ó 201 d. Compruébalo tú mismo mirando en las tablas de mnemónicos, en el Apéndice D del libro.

Para introducir el código en la memoria del ordenador, se usa la orden POKE :

POKE 28672,201

Utiliza la función PEEK para comprobar que el comando se ha ejecutado correctamente.

Una vez estés seguro que la dirección de memoria contiene el código para RET , podemos pasar a ejecutar el programa.

Al teclear RANDOMIZE USR 28672 (ó PRINT USR, o LET L = USR), y ENTER se procederá a:

(1) El microprocesador abandona el modo de ejecución BASIC (abandona el Sistema Operativo).

(2) El Z80 se dirige directamente a la dirección de memoria específica tras USR.

(3) Se comienza a leer e interpretar los códigos que hay en esa dirección de memoria.

(4) Se procesa la operación correspondiente al código.

(5) Si se encuentra una instrucción de retorno (p. e. RET), se abandona el modo de ejecución de Código Máquina, y se vuelve al modo de ejecución BASIC (vuelta al Sistema Operativo).

(6) Si la instrucción no dice lo contrario, se ejecuta el contenido de la siguiente dirección de memoria .

(7) Si se llega a la última dirección de memoria sin haber encontrado ninguna instrucción que le devuelva al BASIC, se pierde el control sobre el Z80. Es necesario desconectar y volver a enchufar.

Todos los programas de código máquina de este libro tienen programas BASIC que cargan los códigos en memoria. Casi siempre de usa la forma RANDOMIZE USR dirección.

La grabación de programa en código máquina de este libro tienen programas BASIC que cargan los códigos en memoria. Casi siempre se usa la forma RANDOMIZE USR dirección

La grabación de programas en código máquina se realiza de la manera siguiente:

(1) Grabando el programa BASIC, que en este libro acompaña a todo listado assembler y que carga los códigos en la memoria del ordenador. Se efectúa tal y como se graban los programas BASIC normales (En realidad, estos programas también son BASIC, aunque tienen en su función algo de código máquina).

(2) Grabando el contenido de un conjunto de direcciones de memoria. Imagina que has realizado un programa que consta de 50 códigos. Estos 50 códigos se almacenarán en 50 direcciones de memoria, cuyo orden es secuencial. Si la primera dirección de memoria usada es la 28672, y cada vez que se introduce un código se hace en la dirección siguiente, habremos ocupado al final 50 direcciones de memoria. En este caso, se ocuparían con códigos las direcciones 28672 a 28721, ambas inclusive.

Es posible grabar el contenido de esas 50 direcciones de memoria con la orden SAVE :

SAVE "nombre" CODE 28672,50

CODE es una función en modo extendido, en la letra I.

El primer argumento especifica la primera dirección de memoria. El segundo, tras la coma, especifica la cantidad de direcciones de memoria que entrarán en la grabación.

En este caso se salvará a cinta magnética los contenidos de las direcciones de memoria 28672 hasta 28722, ésta última no incluida. La razón estriba en que se comienza a grabar los contenidos de las direcciones, comenzando por la indicada antes de la coma, La dirección número 50 será precisamente la 28721.

Este es un detalle que a veces a supuesto la pérdida del contenido de la última dirección, y la posible consecuencia, el erróneo funcionamiento del programa.

Al igual que es posible grabar el contenido de una serie de direcciones, también será posible volverlo a cargar en memoria. La función CODE, conjuntamente con la orden LOAD son las que cumplen este propósito:

LOAD "nombre" CODE 28672,50

Este comando carga la grabación denominada "nombre", y depositará el contenido de la grabación a partir de la dirección de memoria indicada tras CODE. Si la longitud no coincide con la de la grabación, se produce un "R-Tape loading Error". Si se especifica otro valor tras CODE (p. e. 30000) los códigos se comenzarán a cargar a partir de esa dirección (30000).

También puede adoptar la forma:

LOAD "nombre" CODE 28672

En este caso no se indica la longitud de lo grabado. Los códigos se cargan a partir de la dirección de memoria que aparece tras la función CODE, sea cual sea su longitud. Lógicamente, si se excede el máximo (65535) se produce un error de carga.

La tercera forma que puede adoptar LOAD es :

LOAD "nombre" CODE

Este comando carga la información de la cinta a la memoria del aparato. Para ello utiliza exactamente los mismos datos que fueron usados al efectuar la grabación. La información se deposita en el mismo lugar que estaba al grabarse.

Esta última fórmula es seguramente la más sencilla y convincente.

A continuación, el programa DEC - HEX, que transforma un número decimal en su equivalente hexadecimal. El programa es válido para números en el rango 0d a 65535d:

```
 10 REM *** DEC-HEX ***
100 PRINT "DECIMAL" , "HEXADECIMAL"´´
110 INPUT "NUMERO DECIMAL"´D
120 LET D1=D/4096
130 LET D2=(D-(INT D1*4096)}/256
140 LET D3=(D2*256-(INT D2*256})/16
150 LET D4=(D3*16-(INT D3*16))
160 LET D$="0123456789ABCDEF"
170 PRINT D,D$(INT D1+1);D$(INT D2+1);D$(INT D3+1);D$(D4+1)
180 GOTO 110
```

Y este es el programa que transforma un número hexadecimal en su equivalente decimal. Los números hex deben estar en el margen 0000h a FFFFh. Es el caso contrario al programa anterior:

```
100 REM *** HEX-DEC ***
110 PRINT "HEXADECIMAL","DECIMAL"´´
115 POKE 23658,8
120 LET A=10: LET B=11
130 LET C=12: LET D=13
140 LET E=14: LET F=15
150 DEF FN D(H$)=4096*VAL H$(1)+256*VAL H$(2)+16*VAL H$(3)+VAL
    H$(4)
200 INPUT "NUMERO HEXADECIMAL ?"´H$
210 IF LEN H$=2 THEN GO SUB 300
220 IF LEN H$<>4 THEN GO TO 200
230 PRINT H$,FN D(H$)
240 GO TO 200
300 LET H$="00"+H$
310 RETURN
```

3

Estructura de la RAM

La segunda memoria del aparato, la RAM, tiene su comienzo marcado, como ya comentamos, en la dirección de memoria 4000 H /16384 d.

Tiene muchas y variadas funciones. La capacidad total de la RAM es, en un modelo, de 16 K y en el otro, de 48K. Parte de esta memoria es utilizada bien por el microprocesador, o por el sistema operativo, de manera que el número de direcciones de memoria que quedan a la disposición del usuario baja considerablemente. Veremos por qué y cuánto.

1. LA MEMORIA DE PANTALLA

Pertenece en su totalidad a la memoria de lectura y escritura. Esto permite introducir y recoger datos siempre que lo deseemos. Su comienzo está fijado en la dirección de memoria 4000H (16384 D), exactamente donde comienza la memoria de acceso aleatorio. Si introducimos un dato, éste se verá inmediatamente reflejado en la pantalla del televisor o monitor.

Introduce la orden NEW, y tras ella, teclea

POKE 16384,255

La orden NEW asegura que ningún otro programa afectará al nuuestro. El comando POKE introduce en una dirección de memoria (primer argumento) un valor, en el rango 0- 255 (segundo argumento). La orden anterior introduce el valor 255 en la primera dirección de memoria del archivo de pantalla.

En el extremo superior izquierdo de su televisor debe aparecer una pequeña línea negra horizontal. Podrá comprobarlo introduciendo BORDER 0.

La línea negra es el resultado del comando POKE. Ahora debes teclear:

POKE 16384,0

y la pequeña línea horizontal desaparecerá de la pantalla.

La situación de la línea es consecuencia del primer argumento de la orden POKE, que define la dirección de memoria donde se debe introducir el segundo argumento. Así pues, el origen de la línea es el valor introducido, 255.

Recordemos que existe otra forma de contar números en el rango 0-255 usando el formato hexadecimal, que transmite la misma cantidad de información con dos dígitos (00-FF). El sistema hexadecimal nos será útil para trabajar en lenguaje assembler.

Sin embargo, el microprocesador tiene una forma diferente para expresar números en ya mencionado rango. Utiliza para ello el sistema binario. Un byte, compuesto de 8 bits puede contener números del 0 al 255, dependiendo del estado de cada uno de los 8 bits. Antes de proseguir, y si no tiene todavía claro este sistema, vuelva a leer el capítulo dedicado al.tema.

Para aclararnos el misterio de la línea, debemos observar la composición binaria del valor que introduzcamos en la memoria de pantalla. El número 255d o FFH tendrá la siguiente esturctura binaria :

255d = FFH = 11111111 b

Los tres argumentos son iguales, solamente están expresados de forma diferente. La sigla D, H y B indican la base utilizada decimal, hexadecimal y binaria, respectivamente. El valor binario consta de 8 dígitos, pudiendo contener cada uno de ellos el valor 0 ó 1. Para transformar el valor binario en decimal, multiplicaremos cada uno de los bits, comenzando por la derecha por potencias de dos. De esta manera, 11111111 b =

$$1*2^0 + 1*2^1 + 1*2^2 + 1*2^3 + 1*2^4 + 1*2^5 + 1*2^6 + 1*2^7 = 255\ d$$

Una vez demostrado esto, veamos porqué el valor 11111111 b, introducido en la dirección de memoria 16384, da como resultado una línea negra horizontal.

Si introducimos un *bit* con valor 1 en la memoria de pantalla, aparecerá un punto negro en el lugar escogido (tal y como sería si usáramos la orden BASIC PLOT). En cambio, si el bit tuviera el valor 0, "aparecería" un punto en blanco. El significado de "blanco" y "negro" no coincide en este aspecto con el que estamos acostumbrados. Si hablamos de "impresión de un punto en negro", nos referimos a la impresión de un punto con la tinta que en ese momento tenga definida. La "impresión de un punto en blanco" tiene el significado de imprimir un punto "invisible", en el que tinta y fondo tienen el mismo valor. Los puntos "blancos" o "negros" reciben el nombre de *pixels*. Si el punto es "blanco", el pixei tendrá el valor 0 (pixel reset). Un punto "negro" se denomina pixel con valor 1 (pixel set).

La orden POKE 16384,255 introduce directamente 8 pixels con valor 1, pues los 8 bits de los que consta su representación binaria, tienen el valor 1. De esta manera, 8 puntos negros son impresos en la pantalla. Estos 8 pixels forman la línea negra horizontal.

Más adelante, al introducir el valor 0 con la orden POKE, esta misma línea se borraba. Debe-

mos buscar la solución en la composición binaria de 0: 00000000 b =

0*2^1 + 0*2^1 + 0*2^2 + 0*2^3 + 0*2^4 + 0*2^5 + 0*2^6 + 0*2^7
= 0 d

El número binario 0 consta de 8 bits con valor 0. Al trasladarlos a la memoria de pantalla, estos 8 bits se interpretan como 8 pixels de valor 0, 8 puntos "blancos". El resultado visual es el "borrado" de la línea anterior .

Tras un CLS, todas las direcciones de memoria que están incluidas en el archivo de pantalla, exceptuando algunas de la última línea en la que aparece el modo "K" parpadeando, contienen el valor 0.

Precisamente, la rutina de borrado de pantalla de la ROM introduce en cada dirección de memoria del archivo el valor 0.

Para saber qué consecuencia tendrá la introducción de otro valor en la misma posición de la pantalla, debemos buscar la representación binaria del mismo número.

El comando POKE 16384, 170 imprimirá en el extremo superior izquierdo de la pantalla cutro pixels con valor 1 y cuatro con valor 0, alternándose. Este número se expresa en binario de la siguiente manera: 10101010 b

Comprueba tú mismo la igualdad binaria-decimal,, multiplicando cada bit por las potencias de 2.

La memoria de pantalla tiene una longitud de 6144 bytes. Comienza en la dirección 4000H (16384 d) y acaba en la 57FFH (22527 d). El archivo ocupa gran parte de la RAM (6K). En número de pixels, dispondremos de 8*6144 = 49152 pixels (48Kbis).

En baja resolución (uso de caracteres, y no de pixels) tenemos acceso a 32 columnas y 24 filas. En cada una de las intersecciones de fila y columna se puede introducir un carácter. Un caácter es un conjunto de 8 bytes, cuyos pixels representan letras, números o símbolos. En total obtendríamos 32*24 = 768 caracteres ó 768 * 8 bytes = 6144 bytes.

La estructura de la pantalla en alta resolución (acceso a 49152 pixels es compleja: El primer byte del archivo es el primer byte del primer carácter (fila 0, columna 0). Su dirección es, como ya sabemos, 4000H ó 16384 d. La dirección de memoria 16385 corresponde al primer byte del segundo carácter (fila 0, columna 1). Introduce lo siguiente:

```
10 PRINT AT 0,0;"A"
20 POKE 16384,255
30 POKE 16385,255
```

Si ponemos en marcha el programa, el carácter de "A" se imprimirá en la posición (0/0). La sentencia número 20 modifica el primer byte de este carácter, cuyo valor inicial era 0. En pantalla debe aparecer una "A" con una línea negra horizontal sobre ella.

La sentencia número 30 introduce en la dirección 16385 el valor 255. Esta corresponde al primer byte del segundo carácter.

Sin duda adivinarás qué dirección de memoria corresponde al primer byte del tercer carácter. Prueba a introducir aquella dirección un valor cualquiera.

Esta serie continuará hasta el primer byte del último carácter de la fila 0 , el carácter de la columna 31. La información para el primer byte del carácter correspondiente a la fila 0, columna 31 se almacena en la dirección (16384 + 31) = 16415.

Introduciendo la orden POKE 16415,255 observaremos cómo aparece en el extremo superior derecho una línea negra.

Ahora bien, ¿A qué posición de la pantalla corresponde la siguiente dirección de memoria, la 16416?

Sería factible que en esta dirección se almacenase la información para el segundo byte del primer carácter de la pantalla (fila 0, columna 0), el inmediatamente inferior al primer byte del primer carácter (dirección 16384). Los siguientes se ocuparían de los segundos bytes de los caracteres de la primera fila y así hasta el último byte del último carácter (fila 23, columna 31). Lamentablemente este sencillo sistema de estructuración de información no es el usado por el archivo de pantalla.

La dirección 16416 corresponde *al primer byte del primer carácter de la segunda fila*.

Introduce el siguiente programa :

```
10      PRINT AT 1,0; "A"
20      POKE 16416,255
30      POKE 16417,255
```

Este es el ejemplo análogo al primer programa de este apartado, referido esta vez a la segunda fila. Una vez dado RUN, una "A" aparece en la posición (1/0), y justamente sobre ella, ocupando el primer byte de ese carácter, aparece una línea negra. La dirección inmediatamente siguiente, la 16417, corresponde al primer byte del segundo carácter de esta segunda fila.

La dirección de memoria que almacene la información para el primer byte del último carácter de esta segunda fila, será la 16416 + 31 = 16447.

Este sistema proseguirá hasta la octava fila, la fila n. 7 (recuerda que la primera fila era la número cero!)

La dirección de memoria que corresponde al primer byte del primer carácter de la octava fila; es la (16384 + 32 * 7) = 16608. El número de bytes que avanzamos por fila es de 32, y son 7 filas las que separan la primera y la octava.

Escribe tú mismo un programa como los anteriores, referido esta vez a la fila n. 7.

Así mismo, la dirección para el primer byte del último carácter de esta fila, será la (16608 + 31)= 16639.

A partir de aquí, cambia absolutamente toda la serie. La dirección de memoria 16640 no corresponde, como era de esperar, con el primer byte del carácter de la novena fila. Esta dirección almacena la información para el segundo byte del primer carácter de la *primera fila*, el byte inmediatamente inferior al primer byte del primer carácter de la primera fila.

Introduce el siguiente programa :

```
10      POKE 16384,255
20      POKE 16640,255
```

En pantalla aparecen dos líneas, una debajo de otra, en la posición del primer carácter de la primera fila.

Las siguientes 256 direcciones de memoria (8 filas * 32 caracteres) corresponden a los segundo bytes de cada uno de los caracteres de las primeras ocho gfilas. La primera será la 16640, y la última (16640 + 255 =) 16895.

Como debes suponer, la dirección 16896 almacena la información para el tercer byte del primer carácter de la primera fila.

Esto seguirá así, hasta el octavo byte del último carácter de la octava fila, que ocupa la dirección de memoria 18431 d ó 47FFH. Cada carácter tiene 8 bytes, y se usan 256 direcciones de memoria pata cada grupo de bytes. De esta manera, se usan 8 * 256 = 2048 bytes (2K) para el bloqu de caracteres que va del (0/0) al (7/31) . Este bloque corresponde exactamente a un tercio de la pantalla.

El segundo bloque de caracteres de la pantalla comienza en la fila 8, columna 0, y ocupa también otro tercio de la pantalla, hasta la fila 15, columna 31. La organización de los bytes es igual que en el primer tercio.

La dirección de memoria que corresponde al primer byte del primer carácter de este segundo bloque es la 18432 d ó 4800 H.

El segundo bloque también ocupa 2048 bytes (2K) de memoria. Su fin concuerda con la dirección 20749 d ó 4FFFH. El resto de las direcciones corresponden al tercer bloque de caracteres, de igual longitud y estructura que los anteriores.

Este tercio tiene su comienzo en la dirección 20750 d ó 5000H, y su fin en la 22527 d ó 57FFH.

Prueba este programa :

```
10      POKE 22527,255
20      PAUSE 0
```

Tras hacer funcionar el programa podrás observar cómo el último byte de todo el archivo de pantalla, el correspondiente a la fila 23, columna 31, contiene el valor 255. En pantalla aparecerá una línea de 8 pixels en el extremo inferior derecho de la misma.

La sentencia número 20 tiene la función de evitar la aparición del informe "0, OK", que borraría la línea.

La memoria de pantalla ocupa un total de 6K, divididos en 3 grupos de 2K. Cada uno de estos grupos se subdivide en 8 de 256 bytes (1/4 K), correspondiente al primer, segundo, tercero, cuarto, quinto, sexto, septimo y octavo byte de cada uno de los caracteres.

Al cargar una imagen del televisor vía cassete (LOAD " " SCREEN$), vemos cómo van entrando tercio a tercio y byte a byte todo el contenido del archivo de pantalla.

El siguiente programa muestra la estructura de la pantalla:

```
10   FOR F=16384 TO 22527
20   POKE F,255
30   IF F=18431 THEN PRINT AT 4,9;"PRIMER TERCIO":PAUSE 100
40   IF F=20479 THEN PRINT AT 11,9;"SEGUNDO TERCIO":PAUSE 100
50   IF F=22527 THEN PRINT AT 19,9;"TERCER TERCIO":PAUSE 100
60   NEXT F
70   PAUSE 0
```

2. EL ARCHIVO DE ATRIBUTOS

La función del archivo de atributos es el almacenamiento de los colores que aparecen en pantalla. Si la dirección 57FF h /22527 d era la última de la memoria de pantalla, la 5800h /22528 d es la primera del archivo de atributos.

Bajo atributos se debe ententer la capacidad colorística que está a nuestra disposición, que es:

(1) INK o TINTA: 8 colores diferentes pueden ser adoptados como INK o TINTA. Es el color de cada uno de los puntos (o pixels) que forman el carácter.

(2) PAPER o FONDO: Cualquiera de los 8 colores puede ser utilizado como PAPER. Es el color del espacio del carácter no utilizado por INK.

(3) BRIGHT o BRILLO: Esta modalidad permite que el carácter peresente un determinado brillo, o que carezca de él.

(4) FLASH o PARPADEO: Si está activada la opción FLASH, da la sensación que el carácter está parpadeando.

Las opciones OVER o INVERSE no están contenidas en el archivo de pantalla. Es una variable del sistema (ver 4. Variables del sistema) la que almacena la información relativa a estas funciones.

Los colores (atributos) en el ZX Spectrum se usan sólo en formato de baja resolución. Podemos definir un atributo para cada carácter. Un carácter es la denominación general para una letra, un número o un símbolo. 1, F, < , 7, & son diferentes ejemplos para caracteres. Se usan en baja resolución, y en este modo ocupan una casilla de la pantalla, ocupan una intersección de fila y columna. La casilla especificada por una fila y una columna contendrá un carácter. El número de casillas es de 24 filas * 32 columnas = 768 casillas, que pueden contener el mismo número de caracteres. El carácter en sí se almacena en la memoria de pantalla.

De esta manera necesitamos espacio suficiente para almacenar los atributos de esos 768 caracteres. Lo verdaderamente genial es que se ha conseguido estructurar tanto la información, que con un solo byte se definen los atributos para un carácter. La regla de tres no nos engaña: son 768 bytes, almacenados en 768 direcciones de memoria. Si la primera dirección estaba fijada en la 5800H /22528 D, la última será la 5AFFH /23295 D.

La estructuración de la información de los cuatro atributos se basa en el uso de cada uno de los 8 bits que forman el byte.

Dependiendo del estado de esos bits, se interpretará como un atributo diferente. Los 8 bits pueden combinarse de tal forma que se consigan 256 posibilidades diferentes, 256 combinaciones de INK, PAPER, BRIGHT y FLASH.

Para ello se forman 4 agrupaciones de bits. Unas contienen más de un bit, otras solamente uno. Analicemos como sería el contenido típico, el byte típico y normal de una dirección de memoria del archivo de atributos:

0 0 111 000

Precisamente este es el aspecto que presentan todos los bytes del archivo de atributos al ser conectado el ordenador .

De los 4 grupos formados, dos son tres bits y dos de uno. Los bits 0 a 2 (en este caso contienen 000) almacenan la información para la función INK. Estos tres bits se interpretan como 3 cifras binarias.

¿Será casualidad que 3 cifras binarias pueden dar lugar a 8 números diferentes (0-7 decimal), y que existan 8 colores? .

No, no es casualidad el estado de estos tres bits que definen la tinta del carácter. La dirección de memoria que ocupa el byte corresponde a una casilla determinada de la pantalla. Más tarde veremos la relación dirección de memoria-casilla.

El valor del INK al comienzo es siempre 0 (tinta negra), por eso el valor de esos tres bits es cero. Otras combinaciones de los bits darán lugar al resto de los INK:

000 B	---	0 DEC	---	TINTA NEGRA
001 B	---	1 DEC	---	TINTA AZUL
010 B	---	2 DEC	---	TINTA ROJA
011 B	---	3 DEC	---	TINTA VIOLETA
100 B	---	4 DEC	---	TINTA VERDE
101 B	---	5 DEC	---	TINTA CELESTE
110 B	---	6 DEC	---	TINTA AMARILLA
111 B	---	7 DEC	---	TINTA BLANCA

Esta es una de las razones de disponer sólo de 8 colores.

El siguiente grupo de bits es también uno de 3. Los bits número 3, 4 y 5 definen el estado de PAPER o FONDO. Al igual que con INK, disponemos de 3 bits para almacenar 8 valores diferentes y pertenecientes a 8 colores. Al conectar tu ZX Spectrum, PAPER siempre es blanco. El valor correspondiente al color blanco es el número 7 d (Compruébalo mirando la fila superior de teclas), o el 111 B. Al conectar el ordenador los bits n. 3, 4 y 5 del archivo de atributos contienen el valor 1.

El resto de las posibilidades forman otros colores que puede adoptar PAPER :

000 B	---	0 DEC	---	FONDO NEGRO
001 B	---	1 DEC	---	FONDO AZUL
010 B	---	2 DEC	---	FONDO ROJO
011 B	---	3 DEC	---	FONDO VIOLETA
100 B	---	4 DEC	---	FONDO VERDE
101 B	---	5 DEC	---	FONDO CELESTE
110 B	---	6 DEC	---	FONDO AMARILLO
111 B	---	7 DEC	---	FONDO BLANCO

El bit número 6 almacena el estado de BRIGHT o BRILLO. Es un único bit. Si tiene el valor 1, el carácter de la casilla a la que corresponde la dirección de memoria tornará a modo brillante.

Si tiene el valor 0, el estado de brillo quedará desactivado.

0 B	---	0 DEC	---	BRILLO DESACTIVADO
1 B	---	1 DEC	---	BRILLO ACTIVADO

La última función que nosfalta es la de FLASH o PARPADEO. También es un bit único, el

bit número 7, pues al igual que BRIGHT puede adoptar dos modalidades: activado o desactivado. El activado se representa con la cifra 0 (bit con valor 0).

0 B	---	0 DEC	---	PARPADEO DESACTIVADO
1 B	---	1 DEC	---	PARPADEO ACTIVADO

El parpadeo es en sí un contínuo cambio de los valores de INK por los de PAPER. La información de los bit 01, y 2 es intercambiada por la de los bits 3, 4 y 5.

Las funciones "transparencia" (para los cuatro atributos) y "contraste" (para INK y PAPER) se almacenan en otro lugar de la memoria: en las variables del sistema.

Investiga qué atributos proporcionarían los bytes siguientes:

0 1 0 0 0 1 1 1
1 0 1 0 1 0 1 0
0 0 0 0 1 1 1 0
1 1 0 1 1 1 0 1

El contenido decimal de las direcciones de memoria del archivo se puede averiguar de dos maneras. La primera es interpretar el byte como una cifra binaria y cambiarla a base diez.

La segunda es usar la función PEEK. Teclea NEW y ENTER, e introduce:

PRINT PEEK 22528

En pantalla aparece el contenido de la dirección 22528, el valor 56 d. Este es el equivalente al número binario 00111000, que si os acordáis, es el contenido de la información INK 0, PAPER 7, BRIGHT 0, FLASH 0. Estos atributos son los que aparecen al conectar el ordenador.

La relación casilla-dirección de memoria es en el archivo de atributos la siguiente:

La primera dirección de memoria del archivo es la 22528 y que corresponde a la casilla con coordenadas (0/0). Podemos demostrar esto introduciendo en aquella dirección el valor 10010110 b ó 150 d mediante la orden POKE. ¡Házlo!

El valor correspondiente a los atributos INK 6, PAPER 2, BRIGHT / y FLASH 1. La primera casilla de la pantalla está ahora parpadeando, cambiando continuamente de rojo a amarillo, y vice versa.

Siempre que alteremos el contenido de esta dirección de memoria, se produciría un cambio de atributos en la primera casilla de la pantalla.

¿A qué casilla crees que corresponde siguiente dirección de memoria, la 22529? Prueba a ver si coincide con la que pensabas, introduciendo con POKE el valor 150 d.

Efectivamente, la dirección 22529 guarda la información de la casilla con coordenadas (0/1), una columna más a la drecha que la casilla anterior.

Como supondrás, la dirección 22530 es el almacén de la información de la casilla (0/2). Este sistema continúa. Los atributos de la casilla (0/31) se guardan en la dirección de memoria 22559.

Y si pasamos a la siguiente fila, el método sigue siendo el mismo. la dirección 22560 almacena los colores para la casilla (1/0). A medida que incrementamos las dirección, se incrementan también la coordenada de la columna, y cuando se llegue a la 31, la última columna de una línea, se incrementará la coordenada de la fila, y se pondrá a cero la columna.

La última dirección de memoria del archivo corresponde por tanto a la casilla con coordenadas (24/31). Si quieres modificar los atributos de las dos últimas líneas, no olvides poner la instrucción PAUSE 0. La aparición del informe 0,)K borraría los colores.

Este programa introduce en cada dirección de memoria del archivo de atributos un valor aleatorio. La dirección y el valor introducido se imprimen en la pantalla:

```
10   FOR F = 22527 TO 23295
20   LET A = INT (RND*256)
30   POKE F,A
40   PRINT AT 11,0;F;" ";A
50   PAUSE 50
60   PRINT AT 11,0;OVER 1;F;" ";A
70   NEXT F : PAUSE 0
```

El espacio usado para imprimir la dirección y el valor aleatorio es afectado por el POKE, pero los resultados no son visibles, dado que se borra tras la pausa.

Para terminar, un ejemplo a cuya idea se le pueden sacar algún fruto. Pensad que PRINT AT 11,16; "*": POKE 22896,79 equivale plenamente con PRINT INK 7; PAPER 1; BRIGHT 1; FLASH 0;AT 11,16;"*".

3. LA MEMORIA INTERMEDIA DE LA IMPRESORA

En la memoria intermedia de la impresora, también llamada "printer buffer", se almacenan los caracteres a su paso hacia la impresora. Las únicas que usan esta zona son las de conexión directa, como la ZX PRINTER.

Tiene una longitud de 256 bytes (1/4 K), y su comienzo fijado en la dirección 5800 h / 23296 d. Por tanto, su fin estará en la dirección 58FF h /23551 d. Recuerda que la anterior dirección era la últiam del archivo de atributos.

Al recibir una orden LPRINT o LLIST, el ordenador reacciona del siguiente modo:

La información que va a imprimir (una frase o un listado) es analizada. Los bytes que forman sus caracteres son traspadados en grupos de 256, y llenan esta zona de la RAM.

De acuerdo con la impresora, el ordenador va extrayendo la información y enviándosela. Esta la reconoce e interpreta los bytes, imprimiéndolos sobre papel térmico.

Una vez se haya extraído todo el contenjdo del printer buffer, éste se vuelve a "llenar" la información siguiente. El proceso se repite hasta que se termine de cumplir la orden.

El comando COPY recoge, línea por línea, toda la información del archivo de pantalla, y la envia al printer buffer.

Si la impresora y el ordenador no se pueden "poner de acuerdo", esto pasa si la impresora es un modelo convencional y no está adaptada a las normas de Sinclair y del Spectrum, entonces es indispensable el uso de un Interface, que canaliza la información hacia la impresora, haciendo que ésta la entienda.

El comando COPY sólo funciona con impresora de conexión directa, pues su algoritmo, almacenado en ROM, así lo exige. Para tener la orden COPY en impresoras convencionales, hay que hacer un programa en código máquina (esto es código máquina avanzado), o bien adquirir un Interface que ya tenga grabado el programa.

Si no hay una impresora tipo ZX PRINTER conectada, estas direcciones de memoria permanecerán con valor 0, sin ser usadas. Siendo este el caso más corriente, se suelen utilizar estas 256 direcciones de memoria para almacenar pequeños programas o datos.

Estas direcciones de memoria pueden ser utilizadas por el usuario que disponga de una impresora de estas características, a cambio de no usar ni LPRINT, ni LLIST, ni COPY, que ocuparían inmediatamente esta zona de la RAM.

4. LAS VARIABLES DEL SISTEMA

En la zona denominada "Variables del sistema" se almacenan diferentes valores, muy importantes para el microprocesador.

A estos valores se les asigna unos nombres, que tienen relación con sus funciones. Cada uno de estos valores ocupa una dirección de memoria concreta y conocida por el sistema operativo de la ROM.

Estas variables no son conocidas ni por el ordenador, ni por el microprocesador. Son sólo una ayuda para las personas, para que se pueda reconocer su función rápidamente.

Por ejemplo, la variable llamada PROG está ubicada en la dirección de memoria 5C53h y 5C54h / 23635d y 23636d. Estas direcciones contienen la dirección de memoria a partir de la cual se encuentra el PROGrama de BASIC. Este área será analizada más adelante.

No son solamente direcciones vitales, sino también códigos e indicadores los que almacena el microprocesador para ejecutar los programas.

La variable FLAGS2 5C6Ah / 23658 d contiene una serie de indicadores. Uno de ellos controla si está activado el modo "C" o no. Mediante POKE se puede afectar a esta variable y cambiar de "C" a "L " o viceversa en el mismo programa, y sin que nadie pulse CAPS SHIFT-2.

POKE 23658,8 cambia el cursor a modo "C"
POKE 23658,0 cambia el cürsor a modo "L"

La zona de variables del sistema tiene una longitud de 182 bytes. Su comienzo sigue al fin del printer buffer, en la dirección 5C00 h / 23552 d, y se extiende a la 5CB5 h /23733 d. Si el ordenador debe almacenar una dirección, lo hará en base hexadecimal. Para ello necesitará como máximo de dos bytes, uno siendo el byte más significativo, y el otro el menos significativo. La base 16 le proporciona el Z80 un importante ahorro en su memoria.

5. MAPAS DE MICRODRIVE

Esta zona de la memoria se crea en el momento que hay un microdrive conectado y se hace uso de él. El microprocesador necesita de cierto espacio en RAM para el manejo de las unidades de Microdrive.

Se almacenan en esta zona valores necesarios para el manejo de programas o datos.

Comienza exactamente tras las variables del sistema, en la dirección de memoria 5CB6 h / 23734 d. Si se conecta más de un midrodrive, se necesitará más espacio en memoria para manejarlos.

En el ZX Spectrum 16K/48K/+ estándar (sin microdrive o ZX Interface 1) no se llegan a crear los mapas de Microdrive. El ordenador ni siquiera tiene conocimiento de ellos.

Esto significa que se crea un desfase entre los Spectrum con Microdrive y los Spectrum estándar. En el segundo, la siguiente zona de la RAM, los canales de información, ocupa directamente las direcciones de memoria que serían utilizadas por los mapas de microdrive de un Spectrum con este periférico. En el primero, los canales de información ocupan direcciones de memoria más altas que en el segundo.

Al existir este desfase, los comienzos de las siguientes divisiones de la memoria RAM, están

definidos por variables del sistema. Así pues, existe una variable determinada que si están conectados los microdrives, almacenará una dirección diferente a la que sería si no estuvieran conectados. La primera de las dos sería, lógicamente, superior a la segunda.

6. CANALES DE INFORMACION

Los canales de información pueden ser interpretados como una serie de puertas, que se pueden abrir o cerrar, y por las que fluyen mensajes y datos.

El comienzo de esta importante zona está definido por la variable del sistema CHANS (5C4F h y 5C50 /23631 d y 23632 d). Para averiguar el contenido de las variables, que como esta, almacenan direcciones se debe usar la siguiente fórmula:

PRINT PEEK (dirección) + 256 * PEEK (dirección + 1)

El Z80 almacena en primer lugar el byte menos significativo y en segundo lugar el más significativo. Para averiguar el contenido de CHANS, no hay más que introducir los números en su lugar correspondiente:

PRINT PEEK 23631 + 256 * PEEK 23632

y le dará como resultado, si no tiene ni Microdrive ni ZX Interface 1, el valor 23734. Comprueba cómo coincide el comienzo de los inexistentes mapas de microdrive con los canales de información. Si no has comprendido el por qué de la fórmula anterior, no te preocupe, será tratado ampliamente en el capítulo próximo.

Existen cuatro canales, o cuatro puertas diferentes. Una tiene "bisagra doble", se puede abrir a ambos lados. El resto solamente admite una forma de apertura. Los cuatro canales están numerados del 0 al 4, y reciben también una letra.

(1) Canal #0, "K":

Recibe esta letra por ser la primera de "Keyboard" ("Teclado"). Esta es la puerta de doble bisagra. Por una parte, la información se dirige desde el ordenador hasta la pantalla del televisor , concretamente a las dos últimas líneas. Es cuando el Spectrum se comunica con nosotros. Por otra parte, cualquiera de las órdenes o datos que introduzcamos por medio del teclado, hace uso del canal "K".

(2) Canal #1, "R":

Este canal admite solamente salida de información. Es usado normalmente por el propio sistema operativo. La información se envía a otra zona de la RAM, llamada "work-space" o "espacio de trabajo".

(3) Canal #2, "S":

La letra "S" es la primera de "Screen" ("Pantalla"), y por eso cualquier cosa que se vaya a

imprimir en pantalla, excepto en las dos últimas líneas, reservadas al canal #0, "K", utiliza este canal. Así, al usar PRINT, la puerta se "abre" y la información pasa.

(4) Canal #3, "P":
En inglés, "Impresora" recibe el nombre de "Printer". La información dirigida a la impresora está controlada por el canal "P".

Si se usa un Microdrive como archivo de datos, creando un fichero, la información se envía a través de otros canales, que deben ser inicializados con la orden OPEN #. El número de canales disponibles en un Microdrive es de 12.

De todas maneras y en Spectrum estándar, puede usarse también la orden OPEN #:

OPEN #2, "P" cambia la finalidad del canal 2. Lo convierte en canal para impresora. Prueba y verás como PRINT no hace ningún efecto.

PRINT #3; es equivalente a LPRINT. La información debe ser enviada a través del canal #3, el de la impresora.

LIST #3 efectúa un listado del programa en la impresora. Es la misma base que PRINT #3;

PRINT #0; "abc": PAUSE 0. nos permite introducirnos en las dos últimas líneas de la pantalla. El canal #0 puede depositar información en las líneas 22 y 23, imposible de acceder a ellas con la función AT. Es importante escribir la orden PAUSE 0 tras el PRINT, porque si no sería imposible escribir la orden PAUSE 0 tras el PRINT, porque si no sería imposible lograr ver el resultado. El informe 0, OK lo borraría.

Cada uno de los canales ocupa 5 bytes en memoria. En esos 5 bytes se definen direcciones de memoria de las rutinas de entrada/salida (2 bytes para cada una de ellas) y el nombre del canal (1 byte).

Un código determinado (80H/128d) define el fin de la zona de canales de información. En el Spectrum estándar, la longitud es igual a 21 bytes.

7. EL AREA BASIC

Todos los programas, en cualquier lenguaje, deben estar guardados en algún sitio, para que puedan existir.

En el Spectrum, los programas BASIC se almacenan a partir de la dirección de memoria especificada por la variable PROG (5C53 h y 5C54 h /23635 d y 23636 d). Utiliza el método usado anteriormente, y verás que si tu Spectrum carece de Microdrives, la dirección de memoria de comienzo será igual a 23755.

Un programa BASIC de 200 instrucciones ocupará más memoria que uno de 10 instrucciones. La longitud de este área de almacenamiento depende única y exclusivamente del programa en sí.

Es realmente interesante ver cómo se almacena la información de las líneas de un programa. El siguiente programa demuestra cómo está almacenado él mismo:

```
10   LET L = PEEK 23635 + 256 * PEEK 23636
20   PRINT L;TAB 10;PEEK L;TAB 20;CHR$ PEEK L AND PEEK L >32
30   LET L = L +1 : GOTO 20
```

Analicemos los números y caracteres que aparecen en pantalla:

A la izquierda se imprimen las direcciones de memoria que ocupan los códigos que aparecerán en la segunda columna. A la derecha se imprimen los caracteres correspondientes a esos códigos, si se pueden imprimir. Para ello el código tiene que ser superior a 32 d. Los caracteres correspondientes a códigos inferiores 32 d, bien son signos de interrogación o controles de color, bien no son imprimibles. A nosotros nos ínteresa, fundamentalmente, la información que aparece en la segunda y tercera columna.

Los dos primeros códigos son los que corresponden al número de la línea. Este se almacena en base 16, en dos bytes. El primero es el equivalente decimal del byte más significativo, en este caso 00h / 0d. El segundo es el equivalente decimal del byte menos significativo, en este caso 0A h / 10 d. Este número de dos bytes 000A h almacena el número de línea de la primera sentencia del programa. Su equivalente decimal es 10 d. Y efectivamente, el número de línea de la primera sentencia es 10.

Prueba a modificar con POKE el byte menos significativo, introduciendo, por ejemplo, el valor 0. Para ello debes averiguar la dirección de memoria que almacena ese byte.

Los dos códigos siguientes son los equivalentes decimales de la longitud de la I ínea, expresada con dos bytes hexadecimales. En primer lugar aparece el byte menos significativo (al contrario que antes), en este caso su equivalente decimal es 39 d. En segunda posición se encuentra el byte más significativo, que aquí tiene el valor 0. Dado ésto, sabemos que la línea ocupa 39 direcciones de memoria, 39 bytes.

En quinto lugar aparece el código 241, ya su derecha, la palabra LET. Ese código es precisamente el correspondiente al carácter del comando LET. En el sistema Sinclair, los caracteres de cada orden y cada función están definidos, y se les asigna un código determinado. A la orden LET corresponde el código 241. Si en su lugar estuviera la orden REM aparecería el código 234 en una columna, y en la otra el carácter de REM.

Comprueba en el Apéndice cómo el código 76d corresponde con el carácter de L. ¿Una casualidad? Ciertamente que no. El Spectrum almacena los códigos de cada carácter. Estos códigos son interpretados más adelante por el TRADUCTOR BASIC y utilizados consecuentemente por el sistema operativo.

El carácter "=", que aparece tras la "L" en la línea BASIC tiene su correspondencia en la dirección de memoria siguiente. El código 61 d es el que sigue al 76d.

La función PEEK tiene el código 190 d, y aparecen ambos en las dos columnas. Hasta ahora todo marcha estupendamente.

Ahora toca almacenar un número, el 23635 d. En la columna de la derecha puedes ver cada una de las cifras que forman el número. Lo que se almacena es el código de los caracteres de las cifras. ¿Hay diferencia con el almacenamiento de letras ("L") o símbolos ("=")? A simple vista parece que no, pero si investigamos los códigos que aparecen a continuación veremos que sí.

En primer lugar está el código 14, y tras él una serie de caracteres y códigos que no aparecen en la línea BASIC.

En realidad, estos códigos son los que identifican los anteriores con un número. El código 14 aparece siempre tras los códigos de los caracteres del número. Recibe el nombre de "código-número", por su función. Tras este importante código viene la forma que tiene la máquina para almacenar el valor. Para ello se usan 5 bytes y un complicado método.

Observa que para almacenar un número en una línea BASIC, hacen falta 6 direcciones de memoria extra, para el código 14 y los 5 bytes de almacén.

El resto de la línea y del programa debe ser analizado por ti mismo. Busca los códigos y caracteres en el Apéndice y comprueba que todo está en orden.

Sólo hay algo que dudo que pudieras comprender. Para "marcar" el final de cada línea, se hace uso de un código determinado. Es el código 0D h /13 d. Cuando el traductor BASIC se encuentra el código trece, llamado código "ENTER", y no es al definir el valor de un número en los 5 bytes de datos, sabe que la línea ha finalizado, y que debe pasar a la siguiente. El código es de vital importancia, pues el que "separa" literalmente una línea de otra.

El fin del área BASIC se marca con un código diferente, y también importante, el código 80H / 128d.

Cuando el Traductor se encuentra con este código, sabe que el programa ha acabado y que debe ponerse a las órdenes del usuario.

8. EL AREA DE VARIABLES

Justo tras este código 128 d comienza una importante zona de la memoria RAM. De todas maneras, su comienzo está marcado por la variable del sistema VARS (de VARiableS), que ocupa las direcciones de memoria 5C4B h y 5C4C / 23627 d y 23628 d. Al igual que otras muchas zonas, tiene marcado su fin con el código 80H/128d. Tiene la función de almacenar todas las varia-

bles que aparezcan en un programa, sean cuales sean: numéricas, alfanuméricas, conjuntos de números o de caracteres, o las variables de control de bucles.

Solamente decir que a medida que se procesa el programa y se van inicializando variables, esta zona va creciendo en longitud.

Tras un CLEAR, se borran todas las variables, el área tiene 0 direcciones de memoria de longitud.

Prefiero reservar el método utilizado por el ordenador para identificar y no confundir a las variables, hasta el último capítulo del libro, dedicado íntegramente a un programa en código máquina que, analizando este área, extrae todas las variables y las imprime con sus valores respectivos.

9. EL AREA DE EDITAR

Las órdenes directas que se introduzcan en el ordenador, y sobre todo, una línea BASIC que se edite, hacen uso de este área.

La zona de editar tiene su comienzo marcado por la variable del sistema E-LINE, que ocupa las direcciones de memoria 5C59 h y 5C5A / 23641 d. Su fin está determinado por el código 80H/128 d que aparezca a partir de su comienzo.

Se puede interpretar como zona intermedia entre el ordenador y la máquina.

Introduce este programa :

```
10    LET L = PEEK 23641 + 256 * PEEK 23642
20    PRINT L;TAB 10;PEEK L;TAB 20;CHR$ PEEK L AND PEEK L >32
30    LET L = L + 1
40    GOTO 20
```

Este programa inspecciona el área de editar. Para probarlo, introduce una orden larga, que termine con RUN o GOTO 1. Por ejemplo: BEEP 1,0: PLOT 10, 10: PAUSE 10: RUN.

Como podrás ver, todas las órdenes que acababamos de introducir se habían quedado almacenadas en el área.

Es más complicado mostrar un ejemplo editando una línea, porque si tras editar una línea queremos poner en marcha un programa que inspeccione el área de editar (como el anterior) el mismo comando pasa a ocupar ese espacio, y no conseguimos el resultado deseado.

10. EL ESPACIO DE TRABAJO

Si seguimos subiendo en la memoria, encontramos ahora el. espacio de trabajo. Tiene funciones muy diversas y todas dependen del sistema operativo de la ROM. Se utiliza para almacenar y trabajar con ciertos valores especiales.

Es difícil encontrarle una utilidad práctica. Por ahora nos basta con saber que existe, y que se encuentra en la dirección de memoria determinada por la variable del sistema WORKSP (abreviación de WORK SPACE, Espacio de Trabajo), que ocupa las direcciones de memoria 5C61 h y 5C62 h /23649 d y 23650 d.

11. LA PILA DEL CALCULADOR

Esta zona es de uso exclusivo del microprocesador y del sistema operativo. La palabra "pila" tiene aquí el significado de almatén, y no otro. Cuando se realiza una operación, o se ejecuta una expresión matemática, el microprocesador no puede contener todos los valores, y por eso los "guarda" en esta zona.

Se puede comparar con una caja, en la cual podemos meter elementos, pero, eso sí, sólo podemos sacar el de arriba del todo.

El principio y fin de la pila del calculador está determinado por las variables del sistema STKBOT (STacKBOTtom) y STKEND (STacKEND) . La primera ocupa las direcciones de memoria 5C63 h y 5C64 h /23651 d y 23652 d. La segunda las direcciones 5C65 h y 5C66 h /23653 d y 23654 d.

En el capítulo 12, dedicado a los gráficos se hace mención y uso de la pila del calculador.

12. LA MEMORIA DE RESERVA

Es la memoria que le queda al usuario para seguir programando. A medida que se crea un programa y sus variables, la memoria de reserva disminuye. Cuando se acaba, no queda más espacio para trabajar, y el Spectrum emite un zumbido de aviso.

En la versión de 16K, y "gracias" a todas estas divisiones de la RAM (sobre todo a las dos primeras, que se llevan casi 7K), quedan disponibles aproximadamente 8,8 Kbytes.

En la versión de 48K y + se dispone de bastante más memoria, de los 48K de RAM, quedan útiles aproximadamente 41,4 Kbytes. ¡Vaya diferencia!

13. LA PILA DE MAQUINA (Machine Stack)

El microprocesador utiliza este área para almacenar sus propios datos. El mismo posee un sistema que almacena la dirección de memoria, que guarda el último elemento introducido.

Imagina que la pila de máquina está situada en la dirección 32500. Si se introduce un dato en la pila de máquina (existe una instrucción determinada para ello), éste pasaría a ocupar la dirección de memoria 32499. A medida que se introducen datos, el stack va creciendo, pero hacia abajo, siempre ocupando direcciones más bajas. El microprocesador recuerda cada vez la última dirección de memoria.

Si se quisiera sacar un dato, tiene que ser el que esté almacenado en la dirección más baja, pues se usa como elemento de referencia la dirección de memoria almacenada por el Z80.

Todo este sistema y las instrucciones que acompañan a la pila, serán analizadas en un capítulo futuro.

14. LA PILA DE GOSUB

Esta zona la usa el sistema operativo, y es la que permite que el ordenador se "acuerde" del número de sentencia y de orden a la que debe ir un RETURN. en una subrutina.

Su comienzo está fijado por un punto de referencia llamado RAMTOP, que será tratado en el próximo apartado. También crece hacia abajo, siempre ocupando direcciones de memoria más bajas, como la pila de máquina.

El ordenador necesita solamente de tres datos. Uno le dirá el número de orden en la sentencia, y el otro el número de línea de la sentencia a la que el RETURN debe volver.

Introduce este programa, que examina la pila de GOSUB:

```
10   PRINT "PRIMERA SUBRUTINA" : GOSUB 20 : PRINT "FIN": STOP
20   PRINT "SEGUNDA SUBRUTINA" : GOSUB 30: RETURN
30   PRINT "TERCERA SUBRUTINA" : GOSUB 40 : RETURN
40   FOR F= 65365 TO 65357 STEP -1
50   PRINT F,PEEK F
60   NEXT F : RETURN
```

Para 16K, cambia la línea 40 a: FOR F = 32597 TO 32589 STEP -1

El programa realizará las tres subrutinas, y en la última se investigará el contenido de la pila.

En pantalla aparecen unas direcciones de memoria decrecientes, y unos datos. El primero de ellos es un 3, los dos siguientes, un 0 y un 10. El 3 indica que la orden número 3 ("PRINT "FIN" ") la que se debe ejecutar una vez finalizada la subrutina. El número de línea es especificado por los bytes 0 y 10. Son los equivalentes decimales de un número hexadecimal de 4 cifras. El 0 es el byte más significativo, y el 10 el menos significativo. Mediante este sistema se indica al ordenador que el RETURN de la primera subrutina debe volver al tercer comando de la línea 10.

Los otros 6 datos pertenecen al resto de las subrutinas. Lo que se especifica primero es el número de orden (en todos los casos es 3) y después el número de línea (primero 20, y después 30).

15. EL RAMTOP

El RAMTOP (tope de la RAM) es un punto de referencia muy importante de esta memoria. Determina la posición de la pila del GOSUB en memoria, pues éste se encuentra siempre debajo del RAMTOP. Pero sobre todo marca el fin de la RAM que puede ser utilizada por el sistema operativo. Si realizamos un programa, éste podrá expandirse en memoria hasta que se acabe la memoria de reserva, o hasta que se alcance el RAMTOP .

La orden NEW borra el conten ido de la memoria hasta el RAMTOP. Lamentablemente no se pueden colocar programas BASIC por encima del RAMTOP, éstos deben estar en su área.

En la memoria se marca el RAMTOP por dos códigos determinados y por una variable del sistema, llamada RAMTOP (5CB2 h y 5CB3 h /23730 d y 23731 d). Los dos códigos son el 00h y el 3Eh (0d y 62d) . La variable RAMTOP almacena la dirección de memoria que ocupan esos dos códigos.

La posición del RAMTOP en la memoria puede ser modificada con la orden CLEAR dirección. Si se introduce CLEAR 32000, el RAMTOP pasa a ocupar la dirección 32000. Esto trae consigo una alteración en la organización de la memoria. La pila del GOSUB, la pila de máquina y la memoria de reserva son modificadas. Las dos primeras en su posición en memoria, la tercera en su longitud.

Las direcciones de memoria desde el 32000 hasta el fin de la memoria (32767 para 16K y 65535 para 48K) quedan libres a disposición del usuario para almacenar código máquina. Ya veremos cómo. Estas direcciones de la 32000 hasta el fin no serían afectadas por el comando NEW.

La dirección más baja que puede ocupar el RAMTOP es, en un Spectrum estándar, la 23821. Esto equivaldría a quedarse sin memoria de reserva y colocar en posiciones muy bajas la pila de máquina y de GOSUB.

16. LOS GRAFICOS DEFINIDOS POR EL USUARIO

Tienen fijado su comienzo inicialmente a partir del RAMTOP, de manera que una orden NEW no les afecta. Pruébalo y comprueba cómo NEW no borra ninguno de estos gráficos. Introduce ahora:

CLEAR 32767 (16K) a CLEAR 65535 (48K), seguido de NEW

Ahora no existe ningún gráfico definido. NEW los ha borrado. Desconecta y vuelve a enchufar tu Spectrum para que los gráficos vuelvan a aparecer.

Los gráficos definidos son una serie de caracteres (códigos 144 a 164), cuya forma puede ser definida por el propio usuario. Se tiene acceso a 21 gráficos. Para sacarlos en pantalla, cambia a modo "G" y pulsa las teclas de la A a la U. Inicialmente son iguales a los caracteres de mayúscula de las letras A a U.

Cada carácter consta de 8 bytes, es decir de 64 puntos o pixels. El contenido de cada byte se interpreta como un número binario que tiene un equivalente decimal. Este número decimal puede ser introducido mediante POKE. Donde hay que introducir ese número, puede aparecer de dos maneras.

(1) Una dirección de memoria + argumento numérico (de 0 a 255), por ejemplo POKE 32601, valor.

(2) Usando la función USR + argumento alfanumérico + argumento numérico (de 0 a 255), por ejemplo POKE USR "A", valor. El ordenador busca la dirección de memoria del usuario donde se almacene el gráfico definido de la letra A.

El valor que queramos introducir, debe estar en el rango 0 a 255.

Modifiquemos el gráfico de " A":

```
10    POKE USR "A",BIN 00111100
20    POKE USR "A" + 1,BIN 01001110
30    POKE USR "A" + 2,BIN 10011111
40    POKE USR "A" + 3,BIN 10111111
50    POKE USR "A" + 4,BIN 11111111
60    POKE USR "A" + 5,BIN 11011111
70    POKE USR "A" + 6,BIN 01111110
80    POKE USR "A" + 7,BIN 00111100
```

Este programa introduce en las direcciones de memoria correspondientes al gráfico de A una serie de valores que forman un determinado gráfico.

De todas maneras, es más usual escribir directamente los números decimales en vez de los binarios. El programa sería así:

```
10    POKE USR "A" ,60
20    POKE USR "A" + 1,78
30    POKE USR "A" + 2,159
40    POKE USR "A" + 3,191
50    POKE USR "A" + 4,255
60    POKE USR "A" + 5,223
70    POKE USR "A" + 6,126
80    POKE USR "A" + 7,60
```

El "truco" para no tener que escribir tanto la orden POKE "USR"... es colocar los números en DATAS, e irlos leyendo con un bucle FOR-NEXT:

```
10    FOR F = 0 TO 7
20    READ A
30    POKE USR "A" + F, A
40    NEXT F
50    DATA 60, 78, 159, 191, 255, 223, 126, 60
```

La posición en memoria de los gráficos definidos puede ser alterada. La variable del sistema UDG almacena la dirección de comienzo de los 168 bytes (21 gráficos * 8 bytes = 168 bytes = = 168 bytes) . Esta variable ocupa las direcciones 5ClB h y 5C7C h 123675 d y 23676 d.

17. CIERTOS USOS DE LAS VARIABLES DEL SISTEMA

Conociendo las variables de ciertas zonas de la RAM y la función de éstas, pueden ser usadas para, por ejemplo:

(1) Saber cuánta memoria ocupa un programa sin las variables: Para ello, es necesario restar del contenido de la variable del sistema VARS, el contenido de la variable del sistema PROG.

```
                VARS-PROG
PRINT (PEEK 23627+256*PEEK 23628)-(PEEK 23635+256*PEEK 23636)
```

(2) Saber cuánta memoria ocupa un programa con las variables: No hay más que restar del contenido de fa variable del sistema E-LINE, el contenido de la variable del sistema PROG.

E-LINE-PROG
PRINT (PEEK 23641+256*PEEK 23642)-(PEEK 23635+256*PEEK 23636)

(3) Distinguir un 16K de un 48K:

- Se puede hacer leyendo la variable del sistema RAMTOP.

PRINT PEEK 23730+256*PEEK 23731

El 16K da como resultado 32600, el 48K da 65367.

- O leyendo la variable P-RAMT, que almacena la .última dirección de la RAM, la más alta de la memoria.

PRINT PEEK 23732+256*PEEK 23733

El 16K da como resultado 32767, el 48K da 65535.

* * *

Debemos plantearnos cómo y dónde almacenar un programa en código máquina. Existen básicamente dos posibilidades: en líneas REM o en la memoria de reserva. Una tercera posibilidad estriba en el uso de los programas ensambladores. Analicemos cada una de ellas.

1. EN LINEAS REM

Tal y como acabamos de ver, existen zonas de la memoria RAM dedicadas especialmente a un solo fin.

Un programa escrito en lenguaje BASIC se almacena a partir de una dirección de memoria determinada por la variable del sistema PROG . Esta ocupa las direcciones de memoria 5C53h y 5C54h / 23635 d y 23636 d. Podemos conocer en qué dirección de memoria comienza el programa, simplemente leyendo el contenido de PROG. Para ello utilizaremos la orden BASIC PRINT PEEK.

PRINT PEEK 23635+256*PEEK 23636

En pantalla debe aparecer el valor 23755. El usuario del ZX Microdrive habrá advertido que, si ha hecho uso de este periférico, el contenido de la variable PROG es diferente de la que aparece

en pantalla. Esto no se debe a un fallo del ordenador o del periférico. La razón es que el ZX Microdrive y el ZX Interface 1 necesitan de unas direcciones de memoria que, en parte, coinciden con el que sería el comienzo del área BASIC en un Spectrum estándar. Para el que tenga un Microdrive, el área BASIC comienza en direcciones superiores, en la dirección definida por la variable PROG.

A medida que se conectan más drives a la unidad central, éstos necesitan más direcciones de memoria, dejando menos para la programación BASIC.

La razón para todo esto reside en la posibilidad de almacenar código máquina en la zona BASIC. Un programa en código máquina no es más que una serie de códigos o números que son interpretados por el microprocesador.

¿Qué ocurriría si POKEaramos estos códigos directamente en el área BASIC?

Escribe este programa, que consta de una sola línea,

```
10  LET PROG = PEEK 23635 + 256 * PEEK 23636
```

en el que se inicializa la variable con nombre PROG, que contiene el comienzo de la dirección de memoria del mismo programa.

Ahora introduciremos un byte en el área BASIC, un byte que podría ser un código de un programa assembler:

POKE PROG + 10,43 (o cualquier otro número)

y LISTe el programa. El signo de igual (=) ha cambiado. En su lugar está el signo de más (+). Este carácter corresponde al código de 43, que es el número que se POKEó.

Intenta ejecutar ahora el programa, tecleando GO TO 10. El informe "C Nonsense in BASIC 10 : 1 " aparece en la parte inferior de la pantalla. La introducción del código 43 ha supuesto para el ordenador una completa modificación del comando.

En este caso hemos tenido suerte. Si hubiéramos escogido otra dirección podría haberse autodestruido el programa.

Si quieres volver a tener la orden correcta, sin teclear de nuevo toda la línea, haz:

POKE PROG + 10,61

El carácter de igual (=) tiene el código 61. Introduciéndolo en la misma dirección de memoria de antes, se vuelve a modificar su contenido, esta vez a su valor correcto.

Si probamos este mismo sistema con otro tipo de comandos (IF, INPUT, PRINT,...), obten-

dríamos exactamente el mismo resultado negativo. Bueno, con todos no, existe sólo un comando que nos puede servir: REM.

La orden REM tiene la facultad de contener caracteres. Normalmente se usa para introducir comentarios en los programas, que no es más que una serie de caracteres con significado para nosotros. ¿y si se usaran las líneas REM para introducir códigos que tuvieran un significado para el microprocesador?.

Nosotros no entenderíamos los códigos, pero si introducimos la orden RANDOMIZE USR dirección, siendo ésta la que almacena la información, aquellos códigos serían ejecutados.

Probemos este sistema con la instrucción que mejor conocemos, la instrucción RET. El programa assembler constará solamente de esa instrucción. El código objeto de RET es C9h, y el valor equivalente decimal es 201 d.

Desarrollemos la idea de almacenar este código en una línea REM. Como cualquier línea BASIC y como ya dijimos, se estructura de una forma muy peculiar. Seis direcciones de memoria se encargan de almacenarla y mantenerla segura. Tomemos como ejemplo una línea normal :

```
10   REM XXXXXXXX
```

En memoria aparecería así:

```
0 10  10 00        234   88 88 88 88 88 88 88 88        13
```

Los dos primeros bytes determinan el número de línea. Primero aparece el byte más significativo, después el menos significativo. Los dos bytes siguientes almacenan la longitud de la línea a partir del 5 Byte. En primer lugar aparece el byte menos significativo, en segundo lugar el más significativo, al contrario que los dos primeros bytes.

El quinto byte almacena el código del carácter de la orden. En este caso, el código es el 234, que corresponde al carácter de REM . Los 8 bytes siguientes contienen los códigos de los caracteres de "XXXXXXXX".

El último byte corresponde al código 13, de ENTER, que fija el final de la línea.

La orden BASIC a utilizar es POKE. La dirección de memoria la deduciremos de la variable PROG. La línea REM donde almacenaremos el código debe ser la primera del programa.

Hay que asegurarse que se dispone por lo menos del mismo número de caracteres en la línea REM que de los códigos a introducir.

Añade las siguientes líneas al programa y pulsa RUN:

```
20   LET PROG = PEEK 23635 + 256 * PEEK 23636
30   POKE (PROG+5),201
```

Estas líneas localizan la posición del programa en memoria e introducen en la primera dirección de memoria posible (donde reside el primer código de "XXXXXXXXX" el valor 201 d.

La línea tendrá ahora este aspecto:

```
10      REM <>xxxxxxxx
```

Y en memoria, será así:

```
0 10     10 0     234   201 88 88 88 88 88 88 88 88      13
```

El primer código de la línea REM ha cambiado. Su contenido es ahora el 201d, código decimal para la instrucción RET.

Ahora las líneas 20 y 30 pueden ser borradas. Ya han cumplido su función. Para poner en marcha el programa en código máquina, usaremos la orden RANDOMIZE USR (PROG+5), que accede directamente al código 201 d. Este código le hará retornar directamente al BASIC, pero quedará demostrado el almacenaje de código máquina en líneas REM .

Podemos sumarizar el método en 3 pasos:

(1) Escribir una línea REM y el espacio necesario para el código máquina, ocupando con cualquier carácter (por ejemplo, "X"). Este espacio será utilizado más adelante por el código máquina.

(2) Escribir un programa que localice el comienzo del área de BASIC y la dirección de memoria (PROG+5) donde se deben introducir los códigos. El programa BASIC debe introducir el código máquina mediante POKE.

(3) Poner en marcha el código máquina con un RANDOMIZE USR o PRINT USR o LET L + USR.

Este sistema, como cualquier otro trae consigo ventajas y desventajas. Las principales ventajas son las siguientes:

(1) El programa y el código máquina forman una unidad muy compacta. Se pueden grabar y cargar juntos, ahorrando así espacio y memoria.

(2) No hace falta modificar el RAMTOP, ni preocuparse por la longitud del programa BASIC.

Y las desventajas:

(1) Hay que tener cuidado de no teclear el número de línea del REM, pues se borraría todo el código máquina.

(2) Nos encontramos siempre con la obligación de calcular la dirección de memoria del REM En caso de tener dos líneas REM con código máquina, hay que contar con los 6 bytes que todas las líneas ocupan ya de por sí.

En el antecesor del ZX Spectrum, en el ZX 81, la manera más usual de almacenar el código máquina era en líneas REM.

No solamente puedes introducir códigos en líneas REM mediante POKE. Si tecleamos directamente los caracteres que correspondan a los códigos, en este caso, escribiendo directamente una línea REM con el carácter< >. ¡Pruébalo!

Sin embargo, existen una serie de códigos que tienen el mismo carácter, el de interrogación cerrada. Y aunque el carácter es el mismo, el código y el significado no lo es. Y esto llevaría a un CRASH. Por eso es necesario POKEar las direcciones de memoria que contengan un carácter "?" con su contenido decimal correcto, y aunque exteriormente no parece haber habido ningún cambio, éste se ha realizado.

Si hay que introducir un carácter que necesite el cursor en modo K para ser impreso, introduce antes la función THEN. Te aparecerá el cursor en modo K, entonces debe teclear la palabra BASIC que deseaba, y ayudándose de los cursores, borra la función THEN, y sigue introduciendo los códigos.

2. EN EL ESPACIO DE RESERVA

Este método para almacenar códigos se basa en el uso de la memoria o el espacio de reserva de la RAM.

Para ello se escoge una dirección de la memoria suficientemente elevada, a partir de la cual se van almacenando los códigos.

La razón para ello es que ni el programa BASIC, ni las variables de ninguna de las otras zonas de la RAM llegue a "alcanzar" al código máquina y destruirlo.

La mayoría de los programas assembler de este libro están adecuados a la dirección de memoria 7000h o 28672 d, y son válidos, tanto para un 16K, como para un 48K.

Aunque esta dirección de memoria sea relativamente elevada, si deseas incorporar alguna rutina a algún programa tuyo, fijate en la longitud que tenga; quizá debas colocar el código máquina en direcciones superiores.

Almacenando el código máquina a partir de 7000h, quedan poco menos de 5Kb para el usuario, que pueden ser utilizad.os para almacenar BASIC y variables.

Existen diferentes métodos para "proteger" el código máquina frente a crecientes programas o variables. También puede ser destruido si la pila máquina o del GOSUB crece tanto, que llegue a alcanzarla.

La protección se basa en el uso de la orden CLEAR . Esta modifica la posición del RAMTOP , que se encarga de especificar el fin del área BASIC. Si un programa o sus variables alcanza el RAMTOP, el ordenador emite un pitido de advertencia, el mismo pitido que cuando la memoria está llena.

Si se coloca mediante CLEAR 25000 el RAMTOP en esa dirección, y el código máquina a partir de ella, nunca podrá ser alcanzado por los programas.

Otra ventaja que ofrece colocar el código máquina en memoria de reserva, y protegerlo con CLEAR, es que una orden NEW no afectaría al código máquina. El RAMTOP determina la dirección de memoria hasta donde el NEW tiene efecto. Todo lo colocado sobre él, no puede ser borrado por un NEW.

Las pilas de máquina y de GOSUB tampoco podrían afectar al código máquina. Ellas se encuentran bajo el RAMTOP, su punto de referencia, y el código máquina sobre él.

El único método, sin desenchufar, para borrar el código máquina sería con RANDOMIZE USR 0, que equivale a la desconexión.

Este programa es un ejemplo de almacenamiento en memoria de reserva:

```
10   CLEAR 28650 : REM RAMTOP = 28650
20   LET F = 28672: REM 28672: DIRECCION DE INICIO
30   READ A: POKE F,A: LET F = F + 1
40   DATA 201       : REM AQUI MAS CODIGOS SI LOS HUBIERA
```

En este programa se fija como inicio la dirección 28672. Los códigos se almacenan en DATAS, que se van recogiendo e introduciendo en memoria. El programa termina con error "E-OUT OF DATA", una vez que se introduzcan todos los datos. Este error puede ser subsanado programando un bucle que lea el número exacto de datos, y que los introduzca en memoria.

3. PROGRAMAS ENSAMBLADORES

Una tercera posibilidad, menos utilizada que la anterior, se basa en el uso de los programas ensambladores, que permiten introducir directamente los mnemónicos, sin necesidad de usar POKEs ni de preocuparse por los códigos. Los ensambladores pueden identificar directamente los mnemónicos, y buscan los códigos de cada instrucción para introducirlo directamente en memoria.

Si dispones de uno de ellos, familiarizate primero con él y aprovéchalo, no sólo para introducir los listados de este libro, sino también los de tus programas.

De todas maneras, todos los listados assembler del libro traen consigo un programa BASIC que introduce los códigos en memoria de reserva.

4

Más sobre el Código Máquina

1. REGISTROS

El microprocesador tiene la posibilidad de almacenar información en su interior, muy poca, pero algo. Para ello utiliza algo parecido a las variables. Reciben el nombre de *registros*, y también una letra del alfabeto.

La mayor diferencia con una variable es que los registros sólo pueden almacenar numeros entre 0 y 255. Como habrás adivinado, un registro puede almacenar un byte (8 bits) de información.

El registro más importante, el primero de todos, se le ha llamado registro A o Acumulador, precisamente por ser el registro más usado por el Z80, donde almacena y acumula muchas veces bytes. Muchas instrucciones de código máquina sólo pueden trabajar con el acumulador.

También debemos conocer una nueva e importantísima instrucción de assembler. Nos va a permitir introducir en los registros cualquier valor, siempre que esté en el rango permitido.

Es la instrucción LD, abreviatura de LoaD. No tiene nada que ver con el comando BASIC LOAD. LD carga un valor en un registro.

Para cargar en el registro A un valor, por ejemplo 64h / 100d, se usaría la instrucción assembler

```
LD A,64H
```

La mar de fácil ¿no? El valor que aparece tras la coma será el que se introduce en el registro A, en este caso, el número hex 64 o dec 100. Este valor tiene que permanecer en el rango 0 a 255, pues el acumulador tiene sólo 1 byte de 8 bits.

Ahora te preguntarás: "Si, esto es fácil, pero ¿cómo se lo digo yo a la máquina?", y será una acertada pregunta.

Si se lo comunicamos con un mnemónico, no lo entendería, pues no es su idioma. En el Apéndice D encontrarás una tabla en la que debes buscar el mnemónico LD A, N. Esta es una forma

general para escribirlo; N representa un número de 0 a 255. Junto a él debes ver 3E XX. Se trata de dos números en base hexadecimal. La instrucción LD es un típico ejemplo de una instrucción de dos bytes (cada número es un byte) de longitud.

El primer byte (3E h) determina el tipo de instrucción que es, y la operación que se debe realizar: cargar un valor en el acumulador. ¿Pero qué valor?

Precisamente es el byte que aparece a continuación. Recibe el nombre de *byte de dato*, porque su verdadero significado no es ser una instrucción: el microprocesador tomará este byte (XX) como una información unida al anterior. En este caso debe introducir el valor del byte de dato en el registro A. El byte de dato tal y como aparece en las tablas (XX) puede tomar cualquier valor entre 00h y FFh.

Cuando yo aprendí ésto, me hice la siguiente pregunta: ¿Cómo sabe el Z80 qué es una instrucción y qué es un dato?

Para tener claro ésto, hay que hacerse la idea de que el microprocesador toma todos los códigos por instrucciones, excepto si una de ellas le obliga a tomar el/los byte/s siguientes por datos. La interpretación que se le dé a un código depende por tanto de la información que el Z80 haya o no haya recibido antes.

Lo que se encuentra junto al mnemónico LD A, N es el código objeto de la misma instrucción. Es decir, es el código decimal de la instrucción, pero expresado en base 16. El primer byte 3Eh /62d indica al microprocesador que el próximo (en el primer ejemplo el número 64H o 100d) va a ser un dato. Si el valor no es conocido, éste se expresa en el mnemónico con una N y en las tablas con XX.

Mnemónico	Codigo Objeto	Codigo Decimal
LD A,64H	3E64	62,100

Esto son 3 formas de expresar el mismo mensaje: introducir en el acumulador el valor 64H, borrando el anterior .

En BASIC hubiéramos encesitado más de 20 bytes para hacer lo mismo con una variable.

Existen otros muchos registros, y entre éstos están los registros de uso general. También son usados por el microprocesador, aunque no con tanta frecuencia como el acumulador.

Se denominan con otras letras del alfabeto:

B , C , D , E , H y L

La instrucción LD puede ser usada igualmente con *los registros de uso general*. Por supuesto, para cada uno de los registros existe una instrucción con un código diferente. Cualquier valor, en el rango 0 a 255, puede ser introducido en ellos:

Mnemónico	Codigo Objeto	Codigo Decimal
LD B,2AH	062A	6,42
LD C,9CH	0E9C	14,156
LD D,3CH	163C	22,60
LD E,70H	1E70	30,112
LD H,0AH	260A	38,10
LD L,FFH	2EFF	46,255

La potente instrucción LD nos permite igualmente trasladar el contenido de un registro a otro. Existen como instrucciones todas las combinaciones posibles de los registros de uso general con el Acumulador y entre sí. Como ejemplo, sólo mostrar:

LD A, E: traspasa el contenido del registro E al registro A, permaneciendo invariable el registro E.

LD H, D: traspasa el contenido del registro D al registro H, permaneciendo invariable el registro D.

LD B, A: traspasa el contenido del registro A al registro B, permaneciendo invariable el acumulador .

El usar un registro sólo nos permite almacenar números del 0 al 255. ¿Qué hacer para guardar números más grandes?

La respuesta es: usar 2 registros... y precisamente ésta es la solución adoptada por el Z80. Recordarás cómo se almacenaban los números mayores a 255. El usar dos bytes ampl ía nuestras posibilidades hasta el número 65535. Uno de los dos bytes contiene el byte más significativo, y el otro, el byte menos significativo.

Para este fin, el microprocesador hace uso de 2 registros, cada uno de ellos almacenando un byte. Los registros de uso general se unen entre sí en parejas, en *registros dobles*:

Los registros B y C se unen y forman el registro BC.
Los registros D y E se unen y forman el registro DE.
Los registros H y L se unen y forman el registro HL.

Esto no implica que no se pueda trabajar con los registros simples a la vez. Estos forman parte de un registro doble. Si se modifica el valor del registro simple automáticamente se alterará el contenido del registro doble correspondiente.

¿Qué registro almacena el byte más significativo y cuál el menos significativo? Los registros B, D y H almacenan el byte más significativo, y los registros C, E y L el byte menos significativo de los registros dobles BC. DE y HL, respectivamente. Un método para no olvidarse de esto reside en el propio nombre de uno de los registros dobles: el HL. Se ha escogido estas letras, y no las que les seguirían en el alfabeto (G y H -la letra F se reserva para otra función por ser ambas las

iniciales de "HIGH" y de "LOW" ("ALTO" y "BAJO" en castellano). La palabra HIGH representa el byte más significativo, el más importante y por eso, el nombre que se ha asignado al byte más significativo es una "H". La palabra LOW aporta el significado de byte menos significativo, y ésta es la razón para denominar "L" el byte menos significativo del registro doble. Por eso se habla de **bytes altos** (más significativos) y de **bytes bajos** (menos significativos).

Hay que constar que los registros más significativos de los registros dobles (B, D y H) ocupan el primer lugar en el nombre.

Por otra parte, el acumulador puede formar un registro doble, aunque en realidad no se suele usar como tal, con el registro F, del cual hablaremos más adelante.

Imagina que queremos cargar el valor 32767 d en el registro BC. Para ello debemos encontrar primero el equivalente hexadecimal del mismo. Para ello se sigue el siguiente procedimiento, el más rápido que conozco.

(1) Dividir el número decimal entre 256.

(2) La parte entera del resultado es el equivalente decimal del byte más significativo. Hay que guardar el resultado.

(3) Multiplicar la parte entera por 256 y restarla del valor decimal. El resultado será el byte menos significativo.

Este sistema funciona si el valor decimal se mantiene en el rango 0 a 65535.

En el ejemplo de 32767 d, sería así:

(1) 32767/ 256 = 127.99609
(2) 127d = 7FH = byte más significativo
(3) 127d * 256= 32512
32767- 32512 =255 = FFH = byte menos significativo

Así pues, 32767d = 7FFFH

La comprobación puede efectuarse de dos maneras diferentes, bien multiplicando el byte más significativo por 256, y sumar a este resultado el byte menos significativo, o bien multiplicar la primera cifra del número hexadecimal (comenzando por la derecha) por 160, la siguiente por 161, la tercera por 162, y la última por 163, y sumar todos los resu ltados:

(1) 127*256+255=32767d
(2) 255 * 160+ 255* 161 + 255 * 162 + 127 * 163=32767d

Para cargar el valor 7FFFH en el registro doble BC, pueden seguirse dos caminos:

Cargar el registro B con el byte más significativo, y cargar el registro C con el byte menos significativo.

Podemos comprobar que la operación se ha realizado, accediendo a la subrutina con la orden PRINT USR. El valor con que la función USR retorna al BASIC, es precisamente el contenido de este registro doble. Los mnemonicos que formarían el programa adecuado serían:

Mnemónico	Codigo Objeto	Codigo Decimal
LD B,7FH	067F	6,127
LD C,FFH	0EFF	14,255
RET	C9	201

El siguiente programa introduce los códigos en la memoria RAM,

```
10    REM BAJA RAMTOP A 28650
20    CLEAR 28650 : LET T=0
30    REM INTRODUCE CODIGO MAQUINA
40    FOR F=28672 TO 28676
50    READ A: POKE F,A
60    LET T=T+A : NEXT F
70    IF T<>603 THEN PRINT "ERROR EN DATAS": STOP
80    PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
      MAQUINA " : PAUSE 0 : PRINT USR 28672
90    DATA 6 , 127 , 14 , 255 , 201
```

y accede al programa en código máquina. Al retornar se imprime el contenido del registro BC, que como verás, era 32767d. ¡El programa ha funcionado!

Sin embargo, existen unos mnemónicos que permiten introducir directamente en los registros dobles BC, DE y HL los valores que queramos. Tienen la forma siguiente :

Mnemónico	Codigo Objeto	Codigo Decimal
LD BC,NN	01XXXX	1,N,N
LD DE,NN	11XXXX	17,N,N
LD HL,NN	21XXXX	33,N,N

NN representa un número en el rango 0 a 65535. En base 16, éste número tiene 4 cifras y ocupa 2 bytes. En el código decimal deben introducirse los números decimales equivalentes al byte menos significativo y al byte más significativo.

Al igual que anteriormente, podemos realizar un programa. Losmnemónicos serían los siguientes:

Mnemónico	Codigo Objeto	Codigo Decimal
LD BC,7FFFH	017FFF	1,127,255
RET	C9	201

El programa que cargaría los códigos sería el siguiente:

```
10      REM BAJA RAMTOP A 28650
20      CLEAR 28650 : LET T=0
30      REM INTRODUCE CODIGO MAQUINA
40      FOR F=28672 TO 28675
50      READ A: POKE F,A
60      LET T=T+A : NEXT F
70      IF T< >584 THEN PRINT "ERROR EN DATAS" : STOP
80      PRINT "PULSA UNA TECLA PARA PONR EN MARCHA EL CODIGO
        MAQUINA" : PAUSE 0 : PRINT USR 28672
90      DATA 1 , 127 , 255 , 201
```

Este otro programa ocupa un byte menos en memoria. Esto y la mayor rapidez con que se ejecuta la operación son las ventajas para utilizar el segundo método.

Si has introducido el programa y los códigos han cargado satisfactoriamente, observarás que el resultado que aparece en pantalla (65407d), y que es el contenido del registro doble BC, no coincide con el pretendido por nosotros (32767d).

Cambia la línea 90 por la siguiente:

```
90      DATA 1 , 255 , 127 , 201
```

Y pulsa RUN.

Ahora sí que concuerdan ambos resultados. En pantalla aparece el número 32767d. La diferencia entre uno y otro programa es el orden de los bytes. En el primero el byte más significativo (7FH) iba en primer lugar, seguido del byte menos significativo (FFH). En el segundo programa era al contrario. El menos significativo iba en primera posición, seguido del byte más significativo.

Y el segundo programa es el que ha funcionado.

El hecho es que si el Z80 trata números de dos bytes, interpreta el primero de ellos *como el byte menos significativo*, y el segundo *como el byte más significativo*. En el primer ejemplo, el orden era 7FFFH, y por eso el Z80 calculó

$$127 + 255*256 = 65407 \text{ d}$$

cuando en realidad el orden correcto era el del segundo ejemplo FF7FH, en el que el byte menos significativo ocupa la primera posición, y el más importante la segunda

$$255 + 127*256 = 32767 \text{ d}$$

Esta es una de las características irremediables del Z80, siempre almacena los números de dos bytes "al revés". Sólo háy una excepción: el número de línea de las sentencias BASIC tiene como primer byte el más significativo, y como segundo, el menos significativo.

Esta es la razón que si queremos "leer" el contenido de una variable del sistema, que ocupa por ejemplo las direcciones 23635 y 23636, debamos operar así:

PRINT PEEK 23635 + 256*PEEK 23636

La primera dirección (23635) almacena el byte menos significativo o byte bajo (de "low byte"), y la segunda (23636) el byte más significativo o byte alto (de "high byte").

El contenido de la dirección de memoria que almacena el byte alto debe ser multiplicado por 256, ya este resultado se le debe sumar el contenido de la dirección de memoria que almacena el byte menos significativo.

El microprocesador mantiene otra serie de registros, por ejemplo, el registro F. El nombre es inicial de la palabra inglesa FLAGS, cuya mejor traducción dentro de este contexto es la de *Indicadores*. Seis de los ocho bits que tiene el registro almacenan cierta información. Bajo condiciones determinadas puede ser afectado uno o más de los 6 bits. De todos ellos, sólo conoceremos los verdaderamente importantes, que son dos. De estos dos indicadores se puede sacar un gran provecho. Sus nombres y sus funciones se tratarán más adelante.

Un conjunto de registros menos importante para el usuario es el *Grupo de Registros Alternativos* (de Alternative Register Set). Este es un grupo de registros con nombre B', C', D', E', H' y L' (léase B-prima, C-prima, etc.). Al igual que el Grupo de registro de uso general, también pueden agruparse por parejas (BC, DE y HL) . Ei microprocesador sólo puede tener acceso al grupo de registros de uso general y al grupo alternativo, pero nunca a los dos a la vez. En el capítulo dedicado a los Gráficos se habla del acceso al grupo alternativo. Este conjunto se crea, para aumentar el número de lugares de almacenamiento de información del Z80.

También existe el registro A' (Acumulador alternativo) y el F' (Indicador alternativo). El nivel de código máquina usado en este libro no hace uso de ninguno de estos dos últimos registros.

Hablamos de que el microprocesador ten ía un lugar en la RAM para almacenar lo que quisiera. La dirección de memoria de este lugar está contenida dentro de un registro especial, el *Indicador del Stack* registro SP (de Stack Pointer) . Este registro tiene 16 bits, pues debe almacenar una dirección de memoria. Sólo con 8 bits no podría hacerlo. Otro registro llamado PC, es el *Contador de Programa* (de Program Counter) tiene la función de guardar la dirección de memoria cuyo contenido se esté procesando en ese momento. Imagina que el registro PC guarda el valor FFF0H. Una vez haya acabado el microprocesador de ejecutar la instrucción almacenada en dicha dirección de memoria, el Contador de Programa (registro PC) se incrementa en uno, y pasa a valer FFF1 H. Entonces el microprocesador ejecuta la instrucción almacenada en esa dirección. Este registro, al igual que el anterior, es de 16 bits.

Los dos registros siguientes se denominan *registros índice*.

Son los registros IX e IY y tienen ambos 16 bits, es decir, pueden almacenar números en el

rango 0 a 65535. El registro IY es usado a veces por la máquina, y si no se sabe muy bien lo que se hace, es mejor no tocarlo.

Estos registros tienen una característica muy peculiar: Imagina que IX vale 32000. Se tiene un margen de -127 a + 128 direcciones de memoria a las que se puede acceder. El mnemónico

LD A,(IX+32d)

tiene por significado introducir en el registro A el contenido de la dirección de memoria resultante de sumar a IX el número 32. Por eso IX e IY se usan como índices o punteros.

Para terminar, sólo mencionar los registros I (Interrupt) y R (Refresh), los registros de Dirección de Página de Interrupción y de Refresco de Memoria. Son realmente complicados, de manera que no hace falta hablar más sobre ellos.

Recuerda que el mnemónico LD introduce en cualquiera de estos registros, el valor que deseemos.

2. UN LISTADO ASSEMBLER

Al igual que un programa en BASIC tiene una forma determinada, los números de línea a la izquierda, seguidos de los comandos de cada sentencia, un programa escrito en lenguaje ensamblador debe ser expuesto en una forma precisa. Esa exposición se denomina listado, y al estar escrito en el lenguaje ensamblador, se llama listado assembler.

Tomemos como programa ejemplo el utilizado anteriormente, para demostrar la posición de los bytes altos y bajos para el microprocesador. El listado tendría el siguiente aspecto:

```
=======================================================================
Especificaciones : "Programa-ejemplo" para ZX Spectrum 16K/48K
Descripción General : Muestra la codificación correcta para
introducir en el registro BC el valor 32767 d.
Longitud : 4 bytes .
Entrada : Ningún, requerimiento.
Salida : Impresión en pantalla del contenido de BC.
Registros usados : B,C.
Nota: El programa debe ser accedido con PRINT USR.

7000          00100           ORG 70.00H       ;
7000  01FF7F  00110 EJEMP     LD BC,7FFFH      ; BC= 32767 d
7003  C9      00120           RET              ; VUELVE AL BASIC
0000          00130           END              ;

7000 EJEMP
=======================================================================
```

La primera línea expone el nombre del programa y el aparato para el cual está realizado. Junto al apartado "Descripción General" se debe indicar brevemente la función del programa.

El tercer apartado indica la longitud, medida en bytes, del mismo. El punto "Entrada" informa al lector si es necesario efectuar alguna operación antes de acceder al programa, por ejemplo, si se trata del listado de alguna subrutina. El siguiente punto indica los resultados más notables que se producirán una vez ejecutado el programa.

Los registros que se utilizan en el programa se nombran junto al apartado "Registros Usados.". Si hay que hacer una aclaración o nota especial, también debe aparecer en la cabecera del listado.

Lo verdaderamente importante del listado son las líneas que vienen a continuación, que contienen gran cantidad de información para el lector. Como podrás ver, esta parte del listado consta de 6 columnas, cada una de ellas almacena cierta información.

La columna situada más a la izquierda indica qué dirección o direcciones de memoria ocupa la instrucción, cuyo código objeto se encuentra en la segunda columna. El máximo de direcciones de memoria que una instrucción puede ocupar es de 4.

Al igual que un listado BASIC consta de comandos con sus respectivos números de línea, un listado assembler también numera sus instrucciones. La numeración corre a cargo del propio programador. En mis programas assembler empiezo siempre por la línea 00100, y voy incrementándolas siempre de 10 en 10.

La cuarta columna tiene un significado muy especial. En ella se escriben las llamadas "LABELS" o "ETIQUETAS". Tienen la función de dar nombre a un apartado del programa, y si el programador quiere localizarlo, lo hará más rápido si busca una Etiqueta con un significado, que si busca una dirección de memoria. Se puede comparar con las líneas BASIC de la forma "GOTO menu". A medida que vayamos realizando programas assembler, veremos su función y utilidad.

La quinta columna aloja el mnemónico correspondiente al código objeto de la segunda columna. Observa cómo en el código objeto de LD BC, 7FFFH aparece el byte más significativo y el menos significativo en orden inverso (en orden correcto para el Z80) a como aparece en mnemónico.

Si has observado bien, podrás ver que el primero y el último mnemónico carecen de código objeto. En realidad no son instrucciones, sino métodos para comunicarse con el que lee el listado. Por ello reciben el nombre de Pseudo-Mnemónicos. El primero (ORG 7000H) indica al programador la dirección de memoria a partir de la cual está almacenado el programa. ORG es una abreviatura para ORiGen. La dirección de origen se expresa en hexadecimal. El segundo y último pseudo-mnemónico informa del fin de un programa. La traducción para END es FIN.

La última columna va precedida por el signo de punto y coma (;) , y aquí deben escribirse todo tipo de comentarios que el programador estime oportuno (es una zona parecida al REM del lenguaje BASIC) para aclarar la función de los mnemónicos.

Más abajo se indica los valores de cada etiqueta. En el ejemplo, la etiqueta EJEMP tenía el valor 7000H.

La información que será utilizada por el microprocesador será la que aparece en la segunda columna, pues es la única que puede interpretar. El resto sólo es una ayuda al lector del listado.

El que disponga de un programa ensamblador debe introducir el contenido de la quinta columna, y los valores de cada etiqueta. El que no dispone de uno, tendrá que introducir los programas BASIC que acompañan a cada listado assembler, y que contienen los equivalentes decimales de los códigos-objeto, si es que desea hacerlos funcionar. En este caso es el que apareció con anterioridad al listado assembler.

3. UNA LECCION DE ARITMETICA

Una vez conocidos los registros, se nos abren grandes perspectivas en el conocimiento del lenguaje ensamblador.

Una de estas supone el uso de las instrucciones aritméticas que utiliza el microprocesador. Realmente la aritmética no es el fuerte de un microprocesador, pues sólo puede almacenar números en un margen relativamente pequeño, pero es un poderoso elemento de trabajo cuya utilidad la apreciaremos a medida que vayamos realizando programas.

Sumas

El único registro simple con el que el microprocesador puede operar en aritmética es el acumulador, el registro A.

El mnemónico ADD (SUMAR) aporta el significado de sumar a A un valor, el contenido de un registro. Así pues,

ADD A,B

tiene el significado de sumar al acumulador el contenido del registro B. En el caso que A contuviera 5h y que B contuviera 10h, el resultado sería igual a 15h. Este resultado se almacena en el acumulador, borrando el valor que antes contenía (5h).

Este mnemónico tiene un código objeto determinado, el 80H. A partir de ahora deberás buscar en el Apéndice los códigos objeto de cada instrucción que en estas páginas se explique.

La orden BASIC análoga a ADD A, B sería LET A = A + B.

No solamente podemos sumar a A el valor de B. También se puede operar con el resto de los registros de uso general. Cada una de estas instrucciones tendría su propio código objeto.

El mnemónico con la forma

ADD A,N

tiene el significado de sumar al registro A el valor N. Si A fuera igual a 64h y n fuera igual a 7h, el resultado sería 6Bh, que se almacenaría en el Acumulador. El código objeto de este mnemónico consta de 2 bytes. El primero indica al acumulador la operación en sí: Sumar al registro A el valor que venga a continuación, y el segundo aporta el dato, el valor a sumar.

También existe el mnemónico ADD A, A, cuyo significado es fácil de comprender. Al registro A se le suma su propio valor. El resultado es introducido en él mismo.

De los registros dobles de uso general, el microprocesador sólo puede operar con uno, con el registro HL. Esta es otra de las particularidades del Z80. Si el registro simple preferido por él es el Acumulador, el registro doble es el HL.

Podemos sumar con HL el contenido de cualquiera de los otros registros dobles. La instrucción ADD HL, HL también está permitida. El resultado de las sumas siempre se vuelve a almacenar en HL.

Lamentablemente no se pueden sumar registros simples con registros dobles, ni viceversa. Por tanto, una instrucción como ADD BC, A no existe.

El siguiente método supliría esta operación:

- Sumar el contenido de C a A, y traspasar después el contenido del Acumulador al registro C. Lo que habría que hacer ahora sería pasar esta idea a mnemónicos, y después a código objeto. Realiza un listado assembler que muestre el procedimiento. Puedes comprobar el resultado del programa si accedes a él con PRINT U SR. El contenido de BC será impreso en la pantalla. Prueba primero con los siguientes valores:

B = 70H C = 02H A = 18H

Los problemas aparecen cuando la suma de A y C sobrepasa el valor FFh. Prueba y mira lo que ocurre con estos datos:

B = 70H C = 02H A = FFH

Comencemos a analizarlo, explorando el byte menos significativo:

El registro A contiene el valor FFh. El registro C contiene el valor 02h. Una operación ADD A,C se realiza. ¿Qué valor contendrá A tras la operación?

Registro A	1 1 1 1 1 1 1 1
Registro C	0 0 0 0 0 0 1 0

	1

Resultado a almacenar en el Acumulador: ^ 0 0 0 0 0 0 0 1

Con esta operación se sobrpeasa la capacidad del byte. Se ha producido un "arrastre" de una unidad en el último bit (en inglés, se ha producido un "Carry"). Por tanto nos falta un noveno bit para poder expresar el número correctamente, pues el registro A contiene tras la operación el valor 00/00001 b = 1 d = 01 h, y no 100000/0 = 257 d = 101 h.

Aquí es donde entra en juego el registro F, de FLAGS o Indicadores Como dijimos, seis de los 8 bits que lo forman almacenan cierta información. Los indicadores tienen la función de procurar al programador cierta información de lo que está pasando con el acumulador. Uno de ellos, el bit número 0, está dedicado a problemas de esta índole. Este bit almacena el estado del *indicador de arrastre* (CARRY FLAG). Como todos los bits, solamente puede contener dos cifras, el uno o el cero.

El indicador de arrastre tomará el valor 1, si en la última operación aritmética se ha producido un arrastre del último bit, y no se puede expresar ese número con un byte. En cualquier otro caso, contendrá el valor 0.

Este bit de arrastre debe ser sumado al registro B, que almacena el byte más significativo del registro BC. Para ello existe una instrucción especial, llamada ADC A. La operación que realiza se describe como ADD with CARRY (Suma con Indicador de Arrastre) . Cuando el microprocesador lee una instrucción de este tipo, por ejemplo ADC A, B toma el contenido de A y de B y los suma. Pero a este resultado se le añade el estado del indicador de arrastre, que depende del resultado de la última operación. Tras la operación se ajusta el indicador de arrastre al resultado de la suma.

El mnemónico ADC A es muy útil a la hora de sumar valores, cuyos bytes menos significativos pueden sobrepasar el valor 255d. Es el mnemónico que habría que utilizar si quisiéramos resolver el problema anterior.

El listado assembler " ADC BC, A" muestra el programa:
===
Especificaciones : "ADC BC,A" para ZX Spectrum 16K/48K.
Descripción, General : Suma el contenido de un registro doble
con un registro simple.
Longitud : 13 bytes.
Entrada: Ningún requerimiento.

Salida: BC contiene la suma de A+BC.
Registros Usados: A,B y C.
Nota : El programa debe ser accedido con PRINT USR.

```
7000        00100   ORG 7000H      ;
7000  0670  00110   LD B,112       ; B = 70H
7002  0E02  00120   LD C,2         ; C = 02H
7004  3EFF  00130   LD A,255       ; A = FFH
7006    81  00140   ADD A,C        ; A= A +  C
7007   4F   00150   LD C,A         ; C = A
7008  3E00  00170   LD A,0         ; A = 0
700A   88   00180   ADC A,B        ; A = A+B+ARRASTRE
700B   47   00190   LD B,A         ; B = A
700C   C9   00200   RET            ;
0000        00210   END            ;
```

===

Los registros C y A se suman. El resultado de la suma, que reside en el Acumulador es traspasado al registro C. Ahora es necesario operar con el registro B, el más significativo y el indicador de arrastre. Sólo si la suma fue superior a 255, el indicador tomará el valor 1. Sólo se le debe sumar al registro B el valor del indicador, por ello A se inicializa con cero. La instrucción ADC A, B introducirá en A el valor de B, más el del arrastre, más el suyo propio, que es igual a 0. No queda más que traspasar el contenido del Acumulador al registro B. Este es listado BASIC que carga los códigos:

```
 10 REM BAJA RAMTOP A 28650
 20 CLEAR 28650: LET T=0
 30 FOR F=28672 TO 28684
 40 READ A: POKE F,A
 50 LET T=T+A: NEXT F
 60 IF T<>1129 THEN PRINT "ERROR EN DATAS" : STOP
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA": PAUSE 0
 80 PRINT USR 28672
100 DATA 6 , 112 , 14 , 2 , 62 , 255 , 129 , 79 , 62 , 0 , 136 , 71 , 201
```

Si se accede mediante PRINT USR 28672 queda impreso el valor de BC, y se puede comprobar que la suma se ha realizado bien.

Piensa lo que ocurriría si se reemplaza la instrucción de la línea 00180 (ADC A, B) por la instrucción ADD A, B.

La instrucción ADC se puede usar igualmente con el registro doble HL. Su funcionamiento es análogo al de la instrucción ADC A. Observa que en este caso, la instrucción consta de 2 bytes.

Existen varios métodos para sumar valores constantes a los registros. Si se trabaja con el acu-

mulador, la instrucción sería ADD A, constante, pero si se opera registros dobles HL, necesitamos de un "registro-puente", porque un nemónico como ADD HL, constante no existe. Sería necesario hacer un paso más:

LD DE (o BC) , Constante
ADD HL,DE (o BC)

Estos sistemas alterarían el contenido del indicador de arrastre. Si se superaba el bit 7 en un registro simple o el bit 15 en un registro doble, el indicador de arrastre toma el valor 1.

Sin embargo el microprocesador facilita la suma de la constante 1, que al fin y al cabo es la más usada de todas. La instrucción INC tiene el significado de sumar la unidad a un registro.

Al contrario que ADD, INC opera sobre cualquier registro, y no afecta a ningún indicador.

Así pues, INC A aumenta el contenido de Acumulador en 1. Lo mismo ocurre con INC B o INC C, hasta INC L. Si antes de un INC E, el registro E es igual a FFh, tras él, contendrá el valor 00h; esta operación no modifica el contenido del indicador de arrastre.

Esta instrucción puede afectar también a los registros dobles: INC BC, INC DE, INC HL. En estos casos se incrementa.el valor del byte menos significativo, y si sobrepasa el valor 255d, automáticamente es incrementado el byte más significativo, de manera que lo incrementado es el registro doble. Si se trabaja con registros dobles, la instrucción no afecta a ningún indicador. Estas instrucciones INC ocupan solamente un byte en memoria.

Restas

Al igual que en las sumas, el microprocesador solamente puede operar con el registro simple A, y el doble HL.

El mnemónico cuyo significado es el de la resta, recibe el nombre de SUB (de SUBstracción-Substracción). La instrucción SUB A,B contiene el mensaje: Restar el contenido de B al Acumulador. El resultado se almacenará en el registro A. Si A es igual a 80H y b es igual a 10h, el resultado será 70h, que pasará al Acumulador.

La instrucción SUB puede operar con los registros de uso general (registros B, C, D, E, H y L), con el Acumulador mismo (SUB A,A) o con una constante N.

Dado que el único minuendo posible es el registro simple A, se ha adoptado la fórmula de eliminarlo del mnemónico. Las instrucciones toman el siguiente aspecto:

SUB A = SUB A,A
SUB B = SUB A,B
SUB C = SUB A,C
etc

Ambas tienen el mismo significado, sólo es otra manera de expresarlo.

La instrucción SUB A, A introduce en A el valor 0, pero afecta a una serie de indicadores.

Para restar del acumulador una constante N, se usa una orden de dos bytes. El primer byte indica a Z80 ia operación a realizar, restar un valor del Acumulador. El segundo byte aporta la constante a restar. El mnemónico adopta la forma:

SUB N

donde N se mantiene en el rango 0 a 255, al igual que en la suma.

Estas instrucciones SUB afectan a los indicadores. Existen tres casos posibles, cada uno con sus consecuencias.

(1) A = FFH ; B = 01H
SUB A,B
A = FEH ; B = 01H ; INDIC. ARRASTRE = 0

En este caso el sustraendo es menor que el minuendo. El resultado es positivo, y se introduce en el registro A. El indicador de arrastre toma el valor 0, pues no se ha sobrepasado el 255d.

(2) A = 01H ; B = FFH
SUB A,B
A = 02H ; B = FFH ; INDIC.ARRASTRE = 1

El contenido de registro B es superior al contenido del registro A. El resultado de la operación sobrepasa los límites de los bits. Por ello toma el indicador de arrastre el valor 1. El resultado de restar FFh a 01 h se introduce en el Acumulador .

(3) A = FFH ; B = FFH
SUB A,B
A = 0 ; B = FFH ; INDIC. ARRASTRE = 0
INDIC.CERO = 1

El último caso muestra el ejemplo en que minuendo y sustraendo contienen idénticos valores, en este caso FFh. El resultado será igual a cero. No se ha sobrepasado el límite, así que el indicador de arrastre toma el valor 0. En cambio, se ve afectado otro indicador, otro bit del registro F .

Este indicador recibe el nombre de *Indicador de Cero* (ZERO FLAG). Es afectado cuando el resultado de la última operación aritmética es igual a cero. Entonces, el indicador de cero toma el valor 1. En cualquier otro caso, toma el valor 0.

Al igual que existe la instrucción ADC (Add with Carry) , disponemos de SBC (Subtract with Carry) , cuyo significado es Restar con el Indicador de Arrastre.

La orden SBC A, D resta de A el valor D, y de este resultado resta el estado del Indicador de Arrastre. Es decir, si A = 8, D = 4 y el Indicador era igual a 1, una instrucción SBC resta del Acu-

mulador el contenido de D (8 - 4 = 4), y de este resultado, resta el valor del Indicador (4- 1 = = 3). El resultado final se almacena en el registro A. El indicador de arrastre tomará entonces el valor 0, apropiado al resultado.

El único registro doble que permite ser restado es el HL. Pero lamentablemente no existe ninguna instrucción de forma SUB HL, registro. Si se quiere sustraer valores de HL, es indispensable usar la orden SBC. En realidad, la única diferencia con SUB es que se usa el indicador de arrastre, pero si éste tiene el valor 0, SBC y SUB son idénticas.

Los programadores tienen varios métodos para poner a cero el indicador de arrastre. El más usado es XOR A (código AF), instrucción que todavía no hemos visto. A nuestro nivel de conocimientos serviría la instrucción ADD A, 0 que no modifica el contenido de acumulador, pero el del indicador de arrastre.

De esta manera SBC HL, registro resta el contenido del registro que aparece tras la coma del registro que aparece tras la coma del registro HL, y al resultado, le resta el estado del indicador de arrastre. Si antes de la instrucción SBC se pone a cero este indicador, aquella será equivalente a SUB.

No es posible restar el contenido de un registro simple a un registro doble, al igual que pasaba en la suma. Como ejercicio debes realizar un listado assembler cuyo programa realice esta operación.

Al igual que se facilita la suma de la constante 1, restar la unidad del contenido de un registro se realiza mediante un mnemónico especial: DEC. El origen es DECrement (Decrementar), y opera sobre cualquier registro. Ningún Indicador es afectado por esta instrucción. La base es la misma que para INC.

DEC A es igual a SUB A, 1 sólo que no se afecta al indicador y la instrucción ocupa un byte menos en memoria. Las órdenes DEC B, DEC C, DEC D... decrementan los registros respectivos.

DEC puede afectar también a los registros dobles. Si el registro BC contiene 8000H, y se procesa DEC BC, tras la instrucción, el registro contendrá 7FFFh.

Tanto DEC como INC son dos de las instrucciones más potentes de la gama del Z80.

Parte segunda

INSTRUCCIONES

5

Pilas y Subrutinas

1. MANIPULANDO EL STACK

Como ya dijimos, el Stack o pila de máquina es aquella zona de la memoria RAM donde el Z80 almacena un número porque quiere guardarlo o porque le faltan registros para seguir operando.

Una pila es un conjunto de elementos colocados uno sobre otro (recuerda el verbo apilar, amontonar).

El Stack está situado en direcciones muy altas de la memoria. Su comienzo se encuentra bajo el RAMTOP (65367 para 48K y 32599 para 16K).

Debemos imaginar el Stack como una gran caja, donde se van amontonando objetos uno sobre el otro. Por supuesto no son objetos los que allí se introducen, sino números.

Imagina que el Acumulador contiene un valor importante, diganlos el 5. Imagina que la siguiente operación a realizar va a ser una operación aritmética, y que por consiguiente se va a hacer uso del registro A. No se podría realizar dicha operación sin llegar a borrar el contenido previo de A, un valor importante que nosotros queremos guardar.

La solución es almacenar el contenido de A en el Stack, y recuperarlo una vez finalizada la operación :

A CONTIENE VALOR IMPORTANTE (A=5)
ALMACENA EL ACUMULADOR EN EL STACK
REALIZA LA OPERACION ARITMETICA
RECUPERA EL ACUMULADOR DEL STACK (A de nuevo = 5)

Realmente la pila de máquina es un gran elemento de ayuda. Permite guardar y recuperar números, sin que se modifique ningún registro y ningún indicador. El microprocesador "copia" en el Stack el contenido del registro que queramos guardar. Cuando se le pide recuperar los valores, el microprocesador introduce en el registro que queramos el número almacenado en la pila.

De todas maneras, es importante conocer el funcionamiento de la pila de máquina si se quiere sacar buen provecho de ella.

Las complicaciones comienzan si se quiere guardar más de un número. Imagina que los si guientes registros almacenan:

A = 0
B = 128
D = 255

y que sus valores se guardan en el Stack:

ALMACENA EL VALOR DE A EN EL STACK
ALMACENA EL VALOR DE B EN EL STACK
Y ALMACENA EL VALOR DE D EN EL STACK

Antes mencioné que la pila de máquina se podía comparar con una gran caja. El primer número que introducimos es el contenido por el Acumulador, el 0. Este ocupa el fondo de la caja.

El segundo valor que metemos es el contenido por el registro B, el número 128. Al almacenarlo queda "encima" del valor anterior. Y es el último número, el 255 el que queda sobre todos los demás, pues es el último valor que se almacena.

Si se quiere recuperar un valor de la pila de máquina, habrá de ser el que ocupe la posición superior de todos, en este caso, el número 255. Si intentáramos sacar el 128 o el 0, no lo conseguiríamos, seguramente el microprocesador acabaría por volverse loco y tendríamos que desenchufar.

Una vez se haya recuperado un valor, se tiene acceso al que se encontraba "bajo" él, en este caso, el número 128. Hasta que no se saque este valor, no se podrá, ni siquiera intentar, sacar el primero de todos ellos, el 0.

Este sistema de funcionamiento se denomina LIFO (Last In, First Out = Primero Fuera, Ultimo Dentro), y es característico de muchos ordenadores.

La estructura de la pila en la memoria es un tanto anormal. Imagina que el comienzo del Stack en memoria se encuentra en la dirección 32000. Si se introduce un número en la pila, éste pasa a ocupar la dirección de memoria 32000.

¿Qué dirección de memoria crees que ocupará el siguiente número que introduzcamos en el Stack? Normalmente sería almacenado en la dirección 32001, pues normalmente las zonas de almacenamiento de datos, tales como área BASIC o de variables, crecen hacia "arriba", ocupando siempre direcciones de memoria superiores.

La excepción confirma la regla, y son las pilas o Stacks (hay 3 diferentes: de GOSUB, de máquina y de calculador) las zonas de almacenamiento de datos que crecen hacia "abajo", ocupando siempre direcciones de memoria inferiores. En el ejemplo anterior, el siguiente dato no ocuparía la dirección 32001, sino la 31999. y el próximo se guardaría en la dirección de memoria 31998, y el siguiente.en la 31997, y así sucesivamente, hasta que se agotara la memoria de reserva (hay espacio para suficientes números).

Para hacernos una idea de esto, interpretemos el Stack como una caja, colgada del techo de nuestra habitación y en la que podemos introducir, desafiando la ley de la gravedad, tantos objetos como queramos. Al amontonarse uno sobre otro, sólo podremos sacar el último que introdujimos. Si intentamos recuperar otro que no sea éste, se perderá el "equilibrio" y todos los elementos caerán al suelo.

PUSH

El significado del mnemónico PUSH es el de introducir números en el Stack. Supone que DE contiene el número 54321 d, es decir D431H. Esto significa que el registro D guarda el valor INT (54321/256) o 212d / D4h, y que el registro E almacena el valor 54321-256 * INT (54321/256) o 49d / 31h .

Recordarás que existía un registro determinado para el Stack. Su nombre es SP (Stack Pointer), y almacena la dirección de memoria del último número introducido en la pila. A medida que se vayan introduciendo números, el registro SP va disminuyendo, pues siempre se ocupan direcciones de memoria inferiores.

Si se ejecutara una instrucción PUSH DE con los valores anteriores, el microprocesador operaría del siguiente modo:

(1) INTRODUCIR EN LA DIRECCION DE MEMORIA (SP-1) EL BYTE ALTO, EL REGISTRO D: POKE (SP-1), 212 d.

(2) INTRODUCIR EN LA DIRECCION DE MEMORIA (SP-2) EL BYTE BAJO, EL REGISTRO E: POKE (SP-2), 49d.

(3) RESTAR DOS AL REGISTRO SP, PUES DOS DIRECCIONES DE MEMORIA HAN SIDO OCUPADAS. EL REGISTRO SP SIEMPRE "APUNTA" AL BYTE BAJO DEL ULTIMO REGISTRO INTRODUCIDO: SP = SP - 2.

Lamentablemente no es posible introducir registros simples en el Stack, solamente registros dobles:

PUSH AF, BC , DE, HL , IX , IY

Cada uno de estos PUSH tiene su propio código objeto.

Si se quiere almacenar en la pila el valor del Acumulador, hay que utilizar el mnemónico PUSH AF. Este almacena como byte menos significativo el registro F, pero esto es realmente indiferente para nosotros, pues lo que realmente interesa es guardar el registro A.

Recuerda que PUSH no modifica el contenido ni de los registros, ni de los indicadores. PUSH AF puede ser útil si se quiere almacenarel estado de los indicadores en un momento determinado.

POP

La instrucción POP tiene el efecto opuesto a PUSH. Si la función de una era introducir números en el Stack, la función de la otra es recuperar esos valores. Si PUSH sólo introduce el contenido de registros dobles, POP carga en registros dobles los dos bytes que ocupen lo "alto" de la caja (en realidad con la dirección de memoria más baja).

POP hace que el registro SP aumente. Al recuperar números, el Stack Pointer debe contener la dirección de memoria del ahora elemento más alto de la pila. Esta nueva dirección de memoria es ahora superior a la anterior.

Imagina que se ha ejecutado la instrucción PUSH DE, y este registro contiene el valor antes mencionado, D431 h. La instrucción POP DE tendría el siguiente efecto:

(1) INTRODUCIR EN EL REGISTRO BAJO, EN EL REGISTRO E, EL CONTENIDO DE LA DIRECCION DE MEMORIA SP: LET E = PEEK SP (= 49D)

(2) INTRODUCIR EN EL REGISTRO ALTO, EN EL REGISTRO D, EL CONTENIDO DE LA DIRECCION DE MEMORIA (SP + 1): LET D = PEEK (SP + 1) (= 212d)

(3) SUMAR 2 AL REGISTRO SP, DE MANERA QUE SIEMPRE CONTENGA LA DIRECCION DE MEMORIA DEL QUE SERIA PROXIMO BYTE BAJO A POPEAR: LET SP = = SP + 2

No tiene porque existir relación entre el registro que se PUSHea y el que se POPea. Esto ofrece la ventaja de poder, por ejemplo, intercambiar el contenido de dos registros dobles.

El listado assembler "Prog5.1" lo muestra:
===
Especificaciones : "Prog5.1" para ZX Spectrum 16K/48K.
Descripción General : Intercambio de información entre dos registros.
Longitud : 11 bytes.
Entrada : Ningún requerimiento.
Salida: Ningún requerimiento.
Registros Usados : B,C,D,E.

```
7000        00100    ORG 7000H          ;
7000 01E803          00110    LD BC,1000 D      ; BYTE BAJO EN 1. LUGAR !
7003 1131D4          00120    LD DE,54321 D    ; BYTE BAJO EN 1. LUGAR !
7006     C5          00130    PUSH BC           ; GUARDA BC EN STACK
7007     D5          00140    PUSH DE           ; GUARDA DE EN STACK
7008     C1          00150    POP BC            ; INTRODUCE EN BC VALOR DE
7009     D1          00160    POP DE            ; INTRODUCE EN DE VALOR BC
700A    C9           00170    RET               ;
0000       00180     END                ;
```

==

En el programa se le dan a los registros BC y DE los valores 1000 d y 54321 d. Observa como en los códigos correspondientes a los mnemónicos LD aparecen los bytes más significativos y menos significativos en orden invertido.

Después se introduce en el Stack los valores, primero de BC y luego de DE. Al recuperarlos en orden inverso a como se guardaron, se introduce en el registro BC el valor de DE y en éste en valor que antes tenía BC.

A continuación aparece el listado que carga los códigos del programa assembler anterior:

```
 10 REM PROG 5.1
 20 CLEAR 28650: LET T=0
 30 FOR F=28672 TO 28682
 40 READ A: POKE F,A
 50 LET T=T+A: NEXT F
 60 IF T<>1527 THEN PRINT "ERROR EN DATAS" : STOP
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
      MAQUINA": PAUSE 0: PRINT USR 28672
100 DATA 1 , 232 , 3 , 17 , 49 , 212 , 197 , 213 , 193 , 209 , 201
```

Es conveniente asegurarse que un programa dispone de tantos PUSH como POP. Al acceder a un programa en código máquina, el microprocesador almacena en el Stack varios valores que debe utilizar al "volver" al BASIC. Si se quiere ejecutar una orden RET, se necesitarán esos números, pero si en la pila hay algo más, puede ocasionar un CRASH.

El registro SP puede ser manipulado por un programa. Podemos modificar su posición en memoria, colocándolo donde queramos.

El contenido de SP puede ser INCrementado o DECrementado en una rutina, pero debes saber que al final de la misma, y antes de volver a manipular el Stack, el registro SP debe contener una dirección de memoria correcta.

De todas maneras, no hay razón para alterar este registro en los programas que puedas hacer tú ahora. El Stack está.almacenado en lo alto de la memoria, y existe memoria suficiente para cualquier programa.

2. LLAMADAS y RETORNOS

En el lenguaje BASIC es bastante frecuente el introducir subrutinas en los programas. Tienen la función principal de ahorrar memoria al programa y tiempo al realizarlo.

Suelen ser grupos de instrucciones que se repiten mucho, y que por esa razón resulta "económico" el agruparlos en subrutinas y llamarlas cuando se necesiten.

Al final de la subrutina debe aparecer la orden RETURN, que hace que se retorne, que se vuelva hacia la orden siguiente a la que accedió a ella. Los datos necesarios se almacenan en la pila del GOSUB, como ya vimos en el Capítulo Tercero ("Estructura de la RAM").

También es posible disponer de la ventaja de las subrutinas en el lenguaje máquina, sólo que el funcionamiento es algo diferente. En código máquina no se dispone de números de línea, como en BASIC. Las instrucciones se almacenan en direcciones de memoria. Una instrucción puede ocupar una dirección de memoria, o dos, tres o hasta cuatro direcciones. Entonces tendrá la longitud de cuatro bytes.

Si en BASIC es el comando GO SUB, en lenguaje assembler es la instrucción CALL (Llamar). Al igual que en BASIC debe aparecer un número de línea, por ejemplo GO SUB 9900, en ensamblador debe aparecer la dirección de memoria en la que la subrutina debe empezar, por ejemplo CALL 7D00H. Un programa puede contener tantas subrutinas como quiera el programador.

La memoria ROM es en sí una larga serie de subrutinas, que son llamadas por varias rutinas, pertenecientes éstas a diferentes órdenes. Por ejemplo, la rutina del comando RUN utiliza las subrutinas de los comandos CLS, CLEAR, RESTORE y GOTO. Recuerda que RUN equivale a todas estas órdenes.

Alguna de estas subrutinas pueden ser usadas por el programador, aunque hay que saber muy bien qué hace y qué registros deben contener los valores apropiados para realizar la orden. Por ejemplo, se puede acceder a la subrutina de la orden BEEP, para que se realice un sonido. Para ello los registros DE y HL deben almacenar unos números determinados. Esta subrutina es objeto de estudio en el Capítulo Décimoprimero.

La orden que equivale al RETURN del BASIC es en código máquina la instrucción RET. Precisamente la misma que se usa al querer retornar al BASIC, por que en realidad, un programa en código máquina ¿no es una subrutina de un programa en BASIC?

Para introducirnos de lleno en estas dos instrucciones, ha elaborado un programa assembler que suma el contenido de todos los registros. Esta no es su verdadera función, sino la de mostrar el uso de las subrutinas:

Listado Assembler "Prog5.2" :

```
=================================================================================
Especificaciones : "Prog5.2" para ZX Spectrum 16K/48K.
Descripción General  :  Suma de todos los registros. Muestra el uso
 de las subrutinas.
longitud : 38 bytes.
Entrada : Ningún requerimiento.
Salida: Ningún requerimiento.
Registros Usados : A,B,C,D,E,H Y L.
Nota: El programa debe ser accedido con PRINT USR,para que el
resultado pueda ser comprobado .

7000            00100          ORG 7000H       ;
7000   3EF0     00110          LD A,240        ;
7002   01C540   00120          LD BC,16581 D   ;
7005   11A736   00130          LD DE,13991 D   ;
7008   21F31B   00140          LD HL,7155 D    ;
700B   CD0071   00150          CALL SUMA       ;  SUMA HL=HL+DE+BC+A
700E   3E25     00160          LD A,37 D       ;
7010   014700   00170          LD BC,71 D      ;
7013   11450B   00190          LD DE,2885 D    ;
7016   CD0071   00200          CALL SUMA       ;  SUMA HL=HL+DE+BC+A
701A   E5       00210          PUSH HL         ;  PASA EL VALOR DE HL A
701B   C1       00220          POP BC          ;  BC POR MEDIO DE STACK
701C   C9       00230          RET             ;  RETORNA AL BASIC
7100            00240   SUMA   ORG 7100H       ;
7100   81       00250          ADD A,C         ;  A = A + C
7101   4F       00260          LD C,A          ;  C = A
7102   3E00     00270          LD A,0          ;  A = 0
7104   88       00220          ADC A,B         ;  A = A + B + IND.ARR.
7105   47       00230          LD B,A          ;  B = A
7106   09       00240          ADD HL,BC       ;  HL = HL + BC
7107   19       00250          ADD HL,DE       ;  HL = HL + DE
7108   C9       00260          RET             ;  RETORNA DE SUBRUTINA!
0000            00270          END             ;

7100 SUMA
=================================================================================
```

El programa consta de dos partes. La primera es la rutina principal y la segunda la subrutina.

Analicemos la primera:

Los registros A, B, C, D, E, H y L se inicializan con diferentes valores. Todos estos serán sumados al contenido de HL.

La sentencia 00150 realiza la llamada a la subrutina, denominada SUMA. Esta reside en la dirección de memoria 7100h. Podría haber estado ubicada en otra dirección, en la 701 D h o en la 6F00h, simplemente hubiera sido necesario modificar la dirección de memoria expresada tras el CALL. De la subrutina en sí nos hace falta saber que el registro HL retorna de la subrutina con la suma de todos los registros.

El acumulador y 1os registros BC y DE se cargan con una serie de valores, para acceder de nuevo a la subrutina, que sumará el resultado previo y almacenado en HL estos nuevos datos.

El Stack es usado para introducir en BC el resultado final y que guarda HL. De esta manera, y si se accede al programa con PRINT USR se imprimirá el resultado total de la suma. La instrucción nos devuelve al control BASIC.

La subrutina en sí consta de dos apartados. El primero tiene la función de sumar el registro simple A al doble BC. La base del sistema es la misma que la usada en el Capítulo anterior, en el programa "ADC BC,A". Compruébalo tú mismo.

El segundo apartado consiste en sumar los registros BC y DE al registro HL, de tal manera que al ejecutar la orden RET , este registro contiene la suma de todos los demás y el mismo. La orden RET devuelve el control al programa principal.

Esta subrutina es accedida en dos ocasiones. El microprocesador almacena cada vez la dirección de memoria a la que debe retornar.

Al final, el resultado que se debe imprimir en pantalla debe ser igual a la suma de todos los valores contenidos por los registros. Para comprobarlo, sumemos:

$$240 + 16581 + 13991 + 7155 + 37 + 71 + 2885 = 40960$$

Este es el programa BASIC que carga los códigos en memoria:

```
 10 REM PROG 5.2
 20 REM BAJA RAMTOP A 28650
 30 CLEAR 28650: LET T=0
 40 FOR F=28672 TO 28699
 50 READ A: POKE F,A
 60 LET T=T+A: NEXT F
 70 IF T<>2632 THEN PRINT "ERROR EN DATAS 200 O 210" : STOP
 80 LET T=0
 90 FOR F=28928 TO 28936
100 READ A: POKE F,A
110 LET T=T+A: NEXT F
120 IF T<>712 THEN PRINT "ERROR EN DATA 300": STOP
200 DATA 62 , 240 , 1 , 197 , 64 , 17 , 167 , 54 , 33 , 243 , 27 , 205 , 0 , 113 , 62 , 37 ,
       1 , 71 , 0 , 17 , 69 , 11
210 DATA 205 , 0 , 113 , 229 , 193 , 201
```

```
300 DATA 129 , 79 , 62 , 0 , 136 , 71 , 9 , 25 , 201
400 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
      MAQUINA" : PAUSE 0: CLS
410 PRINT "RESULTADO DE LA SUMA : " ; USR 28672
```

Condiciones

Al hablar de condiciones me refiero a la posibilidad de escribir en código máquina instrucciones como

```
IF A < 10 THEN GO SUB 9000
IF A = 10 THEN GO SUB 9100
IF A > 10 THEN GO SUB 9200
```

Por supuesto no tendrán ni mucho menos este aspecto, pero en algo se parecerán. Instrucciones condicionales que comparan números y que toman decisiones a raíz del resultado son realmente útiles. Precisamente el capítulo sobre aritmética va a sernos provechoso.

Las condiciones se basan en el estado de los diferentes indicadores. En este libro serán sólo utilizados y explicados los indicadores de arrastre y de cero, por ser los realmente útiles para la programación.

¿ Cómo podemos saber, si por ejemplo el registro A es superior, igual o inferior al número 10?

La instrucción SUB puede sernos aquí muy útil. Si, operando con SUB, el Acumulador y el número 10 averiguaremos el valor (o una aproximación) al valor de A.

¿ Recuerdas que SUB afectaba a los indicadores de acuerdo al resultado de la sustracción? Existen tres casos posibles:

(1) El contenido del Acumulador es menor que el número restado. El indicador de Arrastre toma el valor 1, pues el resultado fue negativo.

(2) El contenido del Acumulador es igual al número restado. El indicador de Cero toma el valor 1, pues el resultado fue igual a cero.

(3) El contenido del Acumulador es mayor que el número restado. Ni el indicador de Arrastre ni el de Cero se ven afectados.

Gracias a estos dos indicadores ya que existen los mnemónicos CALL condicionales pueden tomarse ese tipo de decisiones:

CALL NN: Llamada incondicional a la dirección de memoria NN.

CALL Z NN: Si la última operación ha puesto a uno el indicador de cero, lo cual quiere decir que la última operación fue igual al valor 0, realiza una llamada a la dirección NN. (CALL if Zero)

CALL NZ NN: Si la última operación ha puesto a cero el indicador de cero, lo cual quiere decir que la última operación fue diferente de cero, realiza una llamada a la dirección NN (CALL if Not Zero) .

CALL C NN: Si la última operación supuso un sobrepasamiento de la unidad byte (faltan bits para expresar el número), realiza una llamada a la dirección NN (CALL if Carry).

CALL NC NN: Si la última operación no supuso un sobrepasamiento del byte, realiza una llamada a la dirección NN (CALL if Not Carry).

Existe otro tipo de llamadas condicionales, usando otros indicadores, pero como ya mencioné me restringiré a los indicadores de arrastre y de cero.

Cada una de estas instrucciones tiene un código objeto determinado, que puede ser encontrado en el Apéndice E del libro, si se busca el mnemónico correspondiente.

Todo esto ya nos podría servir para ayudarnos en nuestro problema inicial. Para saber la relación entre A y 10, no tenemos más que restar el segundo al primero.

```
      SUB A,10
CALL C,A ES MAYOR QUE 10
CALL Z,A ES IGUAL A 10
CALL NC,A ES MENOR QUE 10
```

Estas instrucciones deberían estar insertadas en un programa assembler mayor. Sólo forman un ejemplo de como sería el tomar una decisión condicional con dos números.

El hombre que aparece tras la coma es la etiqueta de aquella parte del programa que alberga las instrucciones para los casos mayor, igual e inferior a 10. Esta etiqueta reemplaza una dirección de memoria.

Las condiciones pueden ser usadas igualmente con la instrucción de RETorno. Al igual que podemos indicar la dirección de memoria de la próxima instrucción a ejecutar con un CALL condicional, podemos decidir el retorno o no retorno hasta que se haya alcanzado una condición determinada. El retorno podrá ser desde una subrutina al programa principal o desde el programa principal al sistema BASIC, pues en ambos casos se utiliza la instrucción RET.

Para saber si la condición se ha cumplido (que por ejemplo A sea igual a 10) se ha de ejecutar una operación aritmética, por ejemplo SUB 10. Esta operación afectará a los indicadores. La decisión será tomada después de acuerdo con su estado. El sistema es el mismo que con los CALL condicionales.

Entre las instrucciones de retorno se encuentran las siguientes:

RET: Retorno incondicional

RET Z: Si la última operación tuvo como resultado el número 0, el Indicador de Cero tomará el valor 1, y se ejecutará un retorno (RET if Zero).

RET NZ: Si la última operación tuvo como resultado un número diferente de 0, el Indicador de Cero tomará el valor 0, y se ejecutará un retorno (RET if Not Zero).

RET C: Si la última operación ocasionó un sobrepasamiento en el número de bits, el Indicador de Arrastre tomará el va lor 1, y se ejecutará un retorno (RET if Carry) .

RET NC: Si la última operación no ocasionó un sobrepasamiento en el número de bits, el Indicador de Arrastre tomará el valor 0, y se ejecutará un retorno (RET if Not Carry).

De todas estas instrucciones, aparte del retorno incondicional, la más usada de todas es RET Z, que permite realizar verdaderos bucles. Imagina el registro A conteniendo el valor 0, y cómo a medida que pasa por un punto determinado del programa, su valor se INCrementa. Una instrucción SUB 100 determinaría si es igual a este valor. La orden RET Z devolvería el programa al BASIC si A es igual a 100. En otro caso se debería ejecutar otra instrucción que dentro de poco conoceremos y que haría que el proceso se repitiese:

```
A = 0
   .....      <------------
A = A + 1            +
SUB 100              +
RET Z                +
------------------------>
```

En esta rutina se ha cometido un grave fallo. No hemos previsto que la instrucción SUB 100 va a alterar ella misma el contenido del Acumulador, sin permitir que llegue a alcanzar el número 100.

Precisamente por ser este último sistema bastante común a la hora de elaborar bucles de este tipo, el lenguaje assembler para el Z80 consta de una instrucción muy potente, que va a suplir a SUB.

La única función de esta nueva instrucción es la de comparar dos números. No afecta el contenido de ningún registro, sólo a los Indicadores. Por ello se denomina CP (de ComPare).

Sería exactamente igual a la instrucción SUB si alterase el contenido de A. SUB introduce en el Acumulador el resultado de la sustracción. CP realiza la sustracción, pero no modifica el valor del Acumulador. Por ello, las únicas consecuencias de CP son las ocasionadas en los Indicadores.

CP es utilizada siempre que se quiera saber si el contenido del registro A ha llegado a un límite determinado. El contenido del Acumulador puede compararse con el contenido de otro registro (CP, B, CP C, CP D...) o con una constante (CP 10d, CP 100d, CP 255d). Si el registro que se quiere comparar no es el A, hay que seguir el siguiente método:

(1) Almacenar el contenido de A si es importante (por ejemplo, en la pila de máquina con PUSH AF).

(2) Introducir en el Acumulador el contenido del registro que se quiera comparar. LD A,B para el registro B, LD A,C para el C...

(3) Realizar la comparación con otro registro o con una constante.

(4) Realizar la instrucción condicional deseada (por ejemplo RET Z).

(5) Recuperar si se almacenó, el contenido previo del Acumulador (si está en la pila de máquina, con POP AF) .

Las llamadas y el Stack

Es muy importante saber manipular el Stack adecuadamente dentro de una subrutina. La razón está clara: el microprocesador consigue acordarse de la dirección de memoria a la que el RET debe dirigirse al abandonar la subrutina, porque antes de acceder a ella guarda en la pila de máquina ese mismo valor.

Cuando se ejecuta una instrucción de retorno, el microprocesador recoge el primer valor de la pila y lo interpreta como la dirección de memoria a la que debe dirigirse.

Por ello tenemos que manejar el Stack con precaución en las subrutinas. Si en una de ellas aparece una instrucción POP BC, para recuperar un valor almacenado antes de la subrutina, ésta introducirá en el registro doble BC la dirección de retorno, y no el valor que nosotros queríamos y que en realidad está "bajo" el primero.

Existen varias soluciones para este problema. Una de ellas es sacrificar un registro doble, pongamos el HL, para que almacene durante la subrutina el valor de la dirección de memoria de retorno. Este valor se introduciría en HL mediante POP HL.

El Stack permanecería entonces libre a nuestros caprichos. Muy importante sería volver a introducir en la pila mediante PUSH HL el valor de la dirección de retorno, antes que el RET se ejecutará. Este sistema tiene la desventaja que durante toda la subrutina tenermos un registro doble inutilizable.

El valor de retorno es indispensable para que el programa y la subrutina puedan funcionar correctamente.

Otro sistema sería el recuperar la dirección de retorno con un registro, que sería usado temporalmente. El contenido del registro se introduciría en una dirección de memoria (más adelante veremos cómo hacer ésto). Tanto este registro doble como el contenido del Stack estarían libres para usarlos.

Al final de la subrutina, antes que el RET se ejecute habría que recuperar de la dirección de memoria anterior el valor de retorno e introducirlo en el Stack.

Existe la posibilidad de modificar la dirección de memoria a la que el RET debe volver, en una subrutina. No hay más que POPear la dirección de retorno introducir la nueva dirección de retorno deseada (pongamos 7D00H) en un registro doble, y PUSHearlo, de manera que el primer valor del Stack sea nuestro nuevo valor .

La orden RET tomará el valor más alto del Stack y se dirigirá a él para seguir operando.

6

Saltos

Cada instrucción de un programa en código máquina está formada por una serie de bytes, de códigos. El número de bytes varía de uno a cuatro. Cada byte se almacena en una dirección de memoria.

El microprocesador mantiene un registro especial, llamado PC (de Program Counter), que almacena la dirección de memoria de la próxima instrucción a ejecutar. Este registro es de 16 bits y por tanto puede almacenar valores entre 0 y 65535, tope máximo de las direcciones de memoria. Imagina el siguiente programa, y observa el contenido de PC en cada instrucción:

```
PC = 28672        LD HL,4000H
PC = 28675        LD A,0
PC = 28677        RET
```

Los códigos de cada instrucción están almacenados a partir de la dirección 28872. Al acceder a este pequeño programa, el registro PC se inicializará con este valor. La primera instrucción consta de 3 bytes, ubicados en las direcciones 28672, 28673 y 28674. El microprocesador lee el contenido del registro PC, que en ese momento es incrementado. Si la instrucción, como en este caso, es de más de 1 byte de longitud, PC se incrementará tantas veces más como bytes tenga la instrucción. El registro PC pasa de valer 28672 a valer 28675. Una vez hecho ésto, se ejecuta la operación correspondiente (cargar en HL el valor 4000H) y se pasa a actualizar de nuevo el contenido de PC, para analizar la próxima instrucción.

PC tiene en este momento el valor 28675, dirección en que comienza la instrucción LD A,0, que consta de 2 bytes, el código de operación y el byte de dato. El registro PC vuelve a incrementarse. A su valor actual se le añade el de la longitud de la instrucción, en bytes. De esta manera, PC toma el valor 28677 y que corresponde al alojamiento de la última instrucción. Una vez hecho ésto, se procede a la operación (cargar 0 en el Acumulador) y al análisis de la instrucción siguiente.

El código de RET consta sólo de un byte. El registro Contador de Programa se incrementa pues en uno, y se retorna al BASIC. Al finalizar el programa, PC vale 28678.

El registro PC se incrementa de tal manera, que siempre contiene la dirección de memoria de la próxima instrucción a ejecutar.

Saltos absolutos

Es posible acceder desde un programa al contenido de PC y modificarlo. Compáralo al GO TO del lenguaje BASIC. Se le puede indicar a qué dirección de memoria debe dirigirse. Para hacer programas que una vez finalizados, volvieran a comenzar, no habría más que indicarle la dirección de memoria de comienzo.

Para todo esto y mucho más se usa un mnemónico llamado JP (de JumP o Salto). Su código objeto está formado por 3 bytes. El primero indica la instrucción de salto en sí. Los dos siguientes son el próximo contenido de PC, la dirección de memoria a la que el microprocesador debe dirigirse.

```
C3 00 70        JP 7000H
```

Observa que la dirección de memoria expresada tras el código C3 tiene el orden lógico (para nosotros) de los bytes invertidos. El byte menos significativo es el que va en primer lugar, seguido del más significativo. En el mnemónico aparece en su orden correcto. Normalmente se suple la dirección de memoria del mnemónico por una etiqueta, de mayor significado para el lector del programa.. Dado que la dirección de memoria a donde se le envía, depende única yexclusivamente de los dos bytes de datos, esta instrucción se denomina de Salto Absoluta.

Tras una orden de este tipo, el programa continuaría a partir de la dirección 7000H. Es muy importante saber calcular (y sobre todo acertar) la dirección de memoria a la que se quiere saltar. Un error de un byte puede ser fatal. Este ejemplo lo muestra:

```
32500     3E06        LD A,6 D
32502     0lFF00      LD BC,255 D
                        . . . .
. . . . . C3F57E      JP 7EF5 H
```

Una serie de instrucciones están alojadas a partir de la dirección 32500 d/ 7FF4 H. Más adelante existe una instrucción que modifica el registro PC a la dirección 7EF5H. Existe un error de un byte, pues 7EF5 h es igual a 32501d. El microprocesador se dirigirá a aquella dirección e interpretará su contenido como un código de operación.

El byte alojado en la dirección 32501 d/ 7EF5 h es 06h, el código objeto de la instrucción LD B, N. El siguiente byte lo interpretará como byte de dato, de manera que introducirá en el acumulador el valor 1. El contenido de la dirección 32503 no será un dato, sino una instrucción. Lo más seguro es que acabara por haber un CRASH.

Solamente con cambiar el byte más significativo por el menos significativo en la instrucción JP se le enviaría a una dirección totalmente diferente: JP F57E = JUMP 62846.

Saltos absolutos condicionales

Al igual que podíamos poner condiciones a la hora de acceder a las subrutinas, podemos hacer lo mismo con los saltos.

Debemos saber que no habrá ningún RET que nos envíe tras la instrucción de Salto, pues entonces sería una subrutina.

IF A = 10 THEN GO TO 9000
u otra condición

Para poder tomar decisiones de acuerdo con determinadas condiciones, tenemos que hacer uso de los indicadores. Entre estos conocemos a los indicadores de Arrastre y de Cero.

Los mnemónicos toman una forma y significado algo diferentes. Entre ellos encontramos los siguientes:

JP NN: Salto absoluto incondicional a la dirección NN.

JP Z NN: Salto absoluto condicional a la dirección NN. La operación se realizará sólo en el caso que el indicador de Cero tenga el valor 1, lo cual indica que el resultado de la última operación fue igual a 0. (JUMP if Zero).

JP NZ NN: Salto absoluto condicional a la dirección NN. La operación se realizará sólo en el caso de que el indicador de Cero tenga el valor 0. El resultado de la última operación fue diferente de cero. (JUMP if Not Zero).

JP C NN: Salto absoluto condicional a la dirección NN. La operación se realizará sólo en el caso de que el indicador de Arrastre sea igual a 1. Indica que la última operación ocasionó un sobrepasamiento en el número de bits. (JUMP if Carry).

JP NC NN: Salto absoluto condicional a la dirección NN. La operación se realizará sólo en el caso de que el indicador de Arrastre sea igual a 0. Esto indica que la última operación no ocasionó un sobrepasamiento en el número de bits. (JUMP if Not Carry).

Estas instrucciones constan de 3 bytes. El primero define la instrucción, y es en cada una de ellas diferente. Los dos bytes siguientes son interpretados como la dirección de memoria a cargar en el registro PC.

La anterior orden BASIC tendría en código máquina su equivalente:-

CP 10
JP Z,A ES IGUAL A 10

DIRECCIONES DE MEMORIA

-128 / 80H	MAXIMO SALTO RELATIVO NEGATIVO
. . . .	
-64 / C0H	
. . . .	
-32 / E0H	
. . . .	
-16 / F0H	
. . . .	
-9 / F7H	
- 8 / F8H	
-7 / F9H	
-6 / FAH	
-5 / FBH	
-4 / FCH	
-3 / FDH	
-2 / FEH	INSTRUCCION JUMP RELATIVE
-1 / FFH	BYTE DE DESPLAZAMIENTO
0 / 00H	
+1 / 01H	
+2 / 02H	
+3 / 03H	
+4 / 04H	
+5 / 05H	
+6 / 06H	
+7 / 07H	
+8 / 08H	
. . . .	
+16 / 10H	
. . . .	
+32 / 20H	
. . . .	
+64 / 40H	
. . . .	
+127 / 7FH	MAXIMO SALTO RELATIVO POSITIVO

La instrucción CP (ComPare) es en estos casos muy útil. En caso que A sea igual a 10, el indicador de Cero tomará el valor 1 y el Contador de Programa se cargará con el contenido de la etiqueta "A ES IGUAL A 10", una dirección de memoria.

Escribe una rutina en código máquina, que una vez accedida no devuelva el control al sistema BASIC hasta que un registro, por ejemplo el B, llegue a valer 0. Su contenido inicial será 0, y deberá ir siendo DECrementada, hasta que llegue a valer de nuevo 0. Es muy sencillo, y sólo hacen falta 7 bytes.

Saltos relativos

La otra posibilidad de ejecutar saltos tiene un nombre muy parecido a JP. Se denomina JR, y proviene de Jump Relative.

Jump Relative modifica también el contenido del registro PC, pero no aporta una dirección determinada que se deba cargar, no aporta una dirección absoluta. Las instrucciones de Salto relativo son de dos bytes de longitud. El primer byte tiene por significado la instrucción JR en sí. El segundo es un byte que expresa un número y que se debe sumar al contenido de PC. Este nuevo contenido es la dirección de memoria de la próxima instrucción a ejecutar. La modificación de PC está por tanto sujeta a su dirección de memoria actual, y al número que se le debe sumar .

El contenido de PC es entonces relativo.

En caso de JP, son dos bytes los que determinan el valor del Contador de Programa, y es totalmente indiferente el contenido que tuviera PC.

Veamos un ejemplo:

```
               LD A,FFH
<----------  JR 04H
+            INC A
+            ADC A,C
+            CP 15H
+-----> CALL Z,SUBRUTINA 1
```

Cuando el microprocesador ejecuta la instrucción JR, el registro PC ya contiene la dirección de memoria de la próxima instrucción a ejecutar. A esta dirección de memoria se le suma un byte de desplazamiento, expresado tras JR, y que tiene el valor 04h.

El contenido actual de PC más el byte de desplazamiento dan la dirección de memoria que alberga la próxima instrucción. En el ejemplo, JR 04h, modifica PC a la dirección de memoria que contiene la instrucción de CALL Z.

Para saber uno mismo la próxima instrucción que será ejecutada tras un salto relativo, se debe contar el desplazamiento indicado a partir de la instrucción siguiente a JR. Nota que no todas las

instrucciones tienen un byte de longitud. Algunas tienen dos o tres o hasta cuatro. En el ejemplo, el desplazamiento es igual a cuatro. Debemos entonces contar cuatro bytes. El primero lo constituiría la instrucción ADC A, C. El segundo y tercero serían la instrucción de comparación, y el cuarto determina el próximo contenido de PC, el comienzo de la instrucción de CALL Z.

Lo realmente potente de JR es la posibilidad de disminuir el contenido de PC. El desplazamiento debe ser entonces negativo. Los números negativos son expresados en lenguaje assembler mediante el complemento a dos.

En realidad el byte de desplazamiento es uno de los usos del método comp:emento a dos. El byte expresado tras JR es un valor en complemento a dos. Lo cual quiere decir, que el máximo salto relativo "hacia adelante" es de + 127 direcciones de memoria.

Si queremos ir "hacia atrás", restando direcciones de memoria a PC, usaremos los números en el rango FFH a 80H (-1 a -128). Veamos un ejemplo:

```
LD DE , 0000H
  LD A , 0lH
    INC A    <---------------
   ADD A , E                +
   LD A , 0                 +
   ADC A,D                  +
     INC A                  +
    JR F8H ----------------->
LD BC , FFFFH
```

Cuando el microprocesador ejecuta la instrucción JR, el registro PC contiene ya la dirección de memoria de la que sería próxima instrucción a ejecutar, en el ejemplo, la dirección de memoria que contiene el primer byte de la instrucción LD BC, FFFFH.

JR suma a PC un número negativo, F8H, de manera que este registro se convierte en:

PC = PC + (-F8H)
PC = PC - 8

puesto que F8H es la representación en complemento a dos de -8. Cuenta tú mismo 8 bytes, comenzando por el que sería el dato de desplazamiento.Recuerda que no todas las instrucciones son de un byte de longitud. Podrás comprobar que la dirección de memoria resultante es la correspondiente a la instrucción INC A.

El máximo salto negativo es de -128 direcciones de memoria, pues trabajamos con complemento a dos (ver Figura Saltos Relativos).

La posibilidad de error es aquí mucho mayor que en los Saltos absolutos. Comprueba qué pasaría si en lugar de poner un desplazamiento negativo de -8, éste es de -9. El significado de los códigos y la función del programa varía totalmente.

Estas son las diferencias más importantes entre JP y JR.

- JP necesita de tres bytes, el primero definiendo la operación, y los dos restantes la dirección de memoria. JR sólo necesita dos bytes. Uno para el código de la operación, y el segundo para el desplazamiento.

- Un programa que sólo utilice instrucciones de salto relativas se puede ubicar en cualquier lugar de la memoria, puesto que la dirección de destino del salto se forma sumando un desplazamiento constante al contenido de PC.

- Con JP se tiene acceso a cualquier dirección de memoria. Con JR hay que mantenerse en el rango -128 a + 127.

En los listados se suele escribir una etiqueta junto al mnemónico, en lugar del desplazamiento, al igual que ocurre con la instrucción JP .

Saltos relativos condicionales

La base es la misma que en los CALL condicionales 0 los JP condicionales. La decisión de ejecutar un salto "hacia adelante" o "hacia atrás" se toma de acuerdo al estado de los indicadores.

Solamente existen los siguientes tipos de salto relativo (DES es una abreviatura para DESplazamiento) :

JR DES: Salto relativo incondicional a la dirección de memoria (PC + DES).

JR Z DES: Si la última operación dio como resultado cero, el Indicador de Cero tomará el valor 1, y se ejecutará un salto a la dirección de memoria (PC + DES) (Jump Relative if Zero DES).

JR NZ DES: Si la última operación dio como resultado un número diferente de cero, el Indicador de Cero tomará el valor 0, y se ejecutará un salto a la dirección de memoria (PC + DES) (Jump Relative if Not Zero DES).

JR C DES: Si la última operación ocasionó un arrastre, el indicador de Arrastre tomará el valor 1, y se ejecutará un salto a la dirección de memoria (PC + DES) (Jump Relative if Carry DES).

JR NC DES: Si la última operación no ocasionó un arrastre, el Indicador de Arrastre tomará el valor 0, y se ejecutará un salto a la dirección de memoria (PC + DES) (Jump Relative if Not Carry DES) .

Cada una de estas instrucciones tienen su propio código objeto, que puede ser encontrado en la tabla Mnemónico-Código Objeto del Apéndice E.

Para mostrar un ejemplo de JP y JR condicionales, podemos dar la solución ahora del problema planteado al fin del apartado ("Saltos Absolutos Condicionales") de este mismo Capítulo.

Se trataba de una pequeña rutina en código máquina que una vez accedida no devuelva el control al sistema BASIC hasta que un registro, por ejemplo el B, llegue a valer 0. Su contenido inicial debe ser 0, y deberá ir siendo DECrementada, hasta que llegue a valer de nuevo 0. Es muy sencillo, y sólo hacen falta 7 bytes.

Esta sería la solución con JP condicional :

```
=======================================================================
Especificaciones : "Prog6.1" para ZX Spectrum 16K / 48K
Descripción General : Ejemplo de JP condicional. El registro B va
decrementándose a partir de 0, y no se ejecuta un RET hasta que
llegue a valer 0 de nuevo.
Longitud : 7 bytes.
Entrada : Ningún requerimiento.
Salida : Valor de B es igual a 0.
Registros Usados : B.

7000        00100   ORG 7000H       ;
7000  0600          00110   LD B,0          ; VALOR INICIAL = 0
7002   05           00120   DEC DEC B       ; B = B - 1
700  C20270         00130   JP NZ,DEC       ; SI B<>0 --> 7002H
7006   C9           00140   RET             ; VUELVE A BASIC
0000        00150   END             ;

7002 DEC
=======================================================================
```

El registro B se inicializa con el valor cero. Junto a la próxima instrucción que decrementa el registro B se encuentra la etiqueta "DEC". Esta será usada por la instrucción de salto.

Si tras la decrementación B es diferente de cero, el Indicador de Cero toma el valor 0, y se produce un salto relativo condicional a la dirección 7002h.

Al comienzo, B pasa de valer 0 a valer 255. En esta operación se verá afectado el indicador de arrastre, pero no el de cero.

El proceso de restar 1 al registro B continuará hasta que alcance el valor, pues en ese caso no se ejecutará la instrucción de salto, si no la siguiente a ésta, el RETorno a BASIC.

Otra solución podía haber sido escribir como línea 00130 la instrucción "RET Z", y como 00140 "JP DEC". El resultado hubiera sido el mismo, pero no se habría utilizado un salto condicional, sino un retorno condicional. Este es el programa BASIC que carga los códigos de "Prog6. 1 ":

```
10 REM PROG 6. 1
20 CLEAR 28650: LET T=0
```

```
 30 FOR F=28672 TO 28678
 40 READ A: POKE F, A
 50 LET T=T+A: NEXT F
 60 IF T<>520 THEN PRINT "ERROR EN DATA": STOP
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA": PAUSE 0
 80 RANDOMIZE USR 28672
100 DATA 6 , 0 , 5 , 194 , 2 , 112 , 201
```

Observa que si resolvermos el mismo problema con saltos relativos, el ahorro de memoria es de un byte. Aparte de esto, la posición que el programa ocupe en memoria es totalmente indiferente:

```
===========================================================================
Especificaciones: "Prog6.2" para ZX Spectrurn 16K / 48K
Descripción General : Como el anterior, pero usando un salto
 relativo condicional.
Longitud: 6 bytes.
Entrada: Ningún requerirniento.
Salida : El registro B vale 0.
Registros Usados : B.

7000      00100            ORG 7000H     ;
7000  0600      00110            LD B,0          ; VALOR INICIAL = 0
7002   05       00120      DEC   DEC B           ; B = B - 1
7003  20FD      00130            JR NZ,DEC       ; SI B<>0 --> DEC
7005   C9       00140            RET             ; VUELVE A BASIC
0000      00150            END           ;

7002 DEC
===========================================================================
```

Compara esto con un bucle FOR-NEXT (FOR B = 256 TO 0 STEP-1). Si introdujeramos instrucciones entre el DEC B y el JRNZ, se ejecutarían tantas veces hasta que B llegara a 0.

Este sistema también puede ser utilizado como pausa en un programa. Se obliga al microprocesador a decrementar un registro hasta que llegue a cero. El tiempo que tarda en hacerlo se interpretará como una pausa. La longitud de la pausa es muy pequeña, menor a milésimas de segundo.
Este es el programa BASIC que carga los códigos:

```
10 REM PROG 6. 2
20 CLEAR 28650: LET T=0
30 FOR F=28672 TO 28677
40 READ A: POKE F, A
50 LET T=T+A: NEXT F
60 IF T< >497 THEN PRINT "ERROR EN DATA" : STOP
```

```
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA" : PAUSE 0
 80 RANDOMIZE USR 28672
100 DATA 6 , 0 , 5 , 32 , 253 , 201
```

Para solucionar esto, se pueden colocar dos bucles JR NZ, controlados por dos registros:

```
=======================================================================
Especificaciones : "Prog6.3" para ZX Spectrum 16K/48K.
Descripción General : Simulacion de pausa con bucles JRNZ.
Longitud : 11 bytes.
Entrada : Ningún Requerimiento.
Salida : Registros B y C con valor 0.
Registros Usados : B , C.

7000        00100            ORG 7000H       ;
7000  0600  00110            LD B , 0        ; VALOR INICIAL B = 0
7002  0E00  00120   BUCLE1   LD C , 0        ; VALOR INICIAL C = 0
7004    0D  00130   BUCLE2   DEC C           ; C = C - 1
7005  20FD  00140            JR NZ,BUCLE2    ; SI C<>0 --> BUCLE2
7007    05  00150            DEC B           ; B = B - 1
7008  20F8  00160            JR NZ,BUCLE1    ; SI B<>0 --> BUCLE1
700A    C9  00170            RET             ;
0000        00180            END             ;

7002 BUCLE1
7004 BUCLE2
=======================================================================
```

Un bucle está en realidad "dentro" del otro. El bucle principal es el controlado por el registro B, que contiene al controlado por el registro C. El equivalente BASIC de este programa sería el siguiente:

```
10 FOR B = 0 TO 255
20 FOR C = 0 TO 255
30 NEXT C
40 NEXT B
```

La pausa resultante es notablemente mayor. Ya continuación aparece el programa BASIC que carga los códigos decimales:

```
10 REM PROG 6.3
20 CLEAR 28650: LET T=0
30 FOR F=28672 TO 28682
40 READ A: POKE F, A
```

```
 50 LET T=T +A: NEXT F
 60 IF T<>804 THEN PRINT "ERROR EN DATA" : STOP
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA": PAUSE 0
 80 RANDOMIZE USR 28672
100 DATA 6 , 0 , 14 , 0 , 13 , 32 , 253 , 5 , 32 , 248 , 201
```

La mayor desventaja de este sistema es que utilizamos dos registros que tienen que ser decrementados.

En código máquina todo tiene arreglo, y para esto existe una instrucción determinada que contiene en sí varias de las anteriores, ocupando sólo dos bytes. Recibe el nombre de DJNZ.

La instrucción DJNZ

El mnemónico es una abreviatura de Decrement, Jump if Not Zero.

La instrucción tiene la forma DJNZ desplazamiento, y es la síntesis de DEC B y JR NZ. Cuando el microprocesador lee esta instrucción, decrementa el contenido del registro B en uno, comprueba si es diferente de cero, y en ese caso, ejecuta un salto relativo a la dirección de memoria resu Itante de sumar a PC el desplazamiento indicado. Se suele utilizar para establecer bucles y pausas.

Es realmente potente, y aporta un ahorro tanto en tiempo de ejecución, como en memoria ocupada. El programa anterior tendría con DJNZ la siguiente forma:

```
===========================================================================
Especificaciones : "Prog6.4" para ZX Spectrum 16K / 48K.
Descripción General : Como "Prog6.2" , pero con DJNZ.
Longitud : 5 bytes .
Entrada : Ningún requerimiento.
Salida : B contiene el valor 0.
Registros Usados : B.

7000      0100              ORG 7000H        ;
7000   0600      0110   PAUSA   LD B , 0          ; VALOR INICIAL = .0
7002   20FE      0120           DJNZ PAUSA        ; Decrernent Jump if Not Zero
7004    C9       0130           RET               ;
0000      0140              END              ;

7002  PAUSA
===========================================================================
```

La instrucción DJNZ PAUSA se envía a sí misma. Ella decrementa B, hasta que sea igual a 0. Entonces volverá al BASIC.

Seguidamente aparece el listado del programa BASIC que carga los códigos decimales del listado assembler anterior:

```
 10 REM PROG 6.4
 20 CLEAR 28650: LET T=0
 30 FOR F=28672 TO 28676
 40 READ A: POKE F, A
 50 LET T=T+A: NEXT F
 60 IF T<>477 THEN PRINT "ERROR EN DATA": STOP
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA": PAUSE 0
 80 RANDOMIZE USR 28672
100 DATA 6 , 0 , 16 , 254 , 201
```

El anterior programa de los 2 bucles entrelazados también puede ser escrito con ayuda de la instrucción DJ NZ. En este caso hay que hacer uso del Stack, para almacenar el valor de B del primer bucle. Una idea más clara la podemos tener con el listado assembler:

```
=======================================================================
Especificaciones : "Prog6.5" para ZX Spectrum 16K / 48K.
Descripción General : Corno "Prog6.3",pero con dos bucles DJNZ.
Longitud : 11 bytes.
Entrada : Ningún requerirniento.
Salida : B con valor 0.
Registros Usados : B.

7000      00100            ORG 7000H       ;
7000  0600       00110             LD B , 0          ;  VALOR INICIAL = 0
7002   C5        00120  BUCLE1 PUSH BC           ;  ALMACENALO EN STACK
7003  0600       00130             LD B , 0          ;  VALOR INICIAL BUCLE 2=0
7005  20FE       00140  BUCLE2 DJNZ, BUCLE2      ;  B=B-1 HASTA QUE B=0
7007   C1        00150             POP BC            ;  RECUPERA BC DE STACK
7008  20F8       00160             DJNZ, BUCLE1      ;  B=B-1; SI B<>0 --> BUCLE1
700A   C9        00170             RET               ;
0000             00180             END               ;

7002  BUCLE1
7005  BUCLE2
=======================================================================
```

La gran diferencia entre éste y el programa "Prog6.3", es que éste hace uso solamente de un registro, el B. Dado que se usa dos veces, el valor del bucle principal se almacena en el Stack mientras se ejecuta el bucle.secundario. Una vez terminado, se recupera y decrementa. Hasta que el valor del registro B del bucle principal no haya alcanzado el cero, el proceso se repetirá.

Este es el listado BASIC correspondiente al programa assembler "Prog6.5":

```
 10 REM PROG 6.5
 20 CLEAR 28650: LET T=0
 30 FOR F=28672 TO 28682
 40 READ A: POKE F,A
 50 LET T=T+A: NEXT F
 60 IF T<>1137 THEN PRINT "ERROR EN DATA": STOP
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
      MAQUINA" : PAUSE 0
 80 RANDOMIZE USR 28672
100 DATA 6 , 0 , 197 , 6 , 0 , 16 , 254 , 193 , 16 , 248 , 201
```

7

Modos de Direccionamiento

Este capítulo tiene la función de familiarizarte con los registros y sus usos. Hablaremos de diversos modos de direccionamiento del microprocesador Z80.

Un modo de direccíonamiento es un método por el cual el programador, que eres tú, puede acceder a un dato por medio de una instrucción. Los datos pueden ser constantes numéricas, los contenidos de otros registros, o de direcciones de memoria. Para cada uno de estos casos existe un mnemónico y un código diferente. Cada modo de direccionamiento recibe un nombre determinado, pero lo importante es comprender la función de cada uno de ellos, no su denominación.

- El ejemplo más sencillo es el que introduce en un registro simple un número: LD A, FFH (código 3EFF).

El código de la instrucción consta de dos bytes. El primero de ellos determina la operación a realizar, que es cargar un valor en un registro. Para introducir un número en el Acumulador, el código es el 3EH. Cuando el microprocesador "lee" este código, es informado de lo que tiene que hacer. El dato a introducir en el registro se encuentra a continuación. Como ya hablamos, este recibe el nombre de byte de dato, en contraposición al primero, que es el byte de operación.

En el mnemónico, registro y dato son separados por una coma, para que no pueda haber lugar a ninguna duda.

El dato se carga inmediatamente en el registro, sin necesidad de hacer uso de la memoria u otro componente del aparato, y por ello, esta forma de tratar datos recibe el nombre de **direccionamiento inmediato**. Cualquier registro de uso general más el Acumulador pueden ser afectados por el direccionamiento inmediato.

Por supuesto, los datos sólo pueden mantenerse en el rango 0 a 255 decimal, máxima capacidad de un byte de 8 bits. A la hora de escribir el código objeto, el dato debe expresarse en sistema hexadecimal, en el rango 00 a FFH.

El modo anterior de direccionamiento afecta solamente a un registro simple. Sin embargo, hemos utilizado otro sistema para introducir datos en dos registros a la vez.

Los registros deben agruparse de dos en dos. Esto lo pueden hacer los registros de uso general. El registro B forma con el C el registro doble BC. Lo mismo ocurre con D y E, que forman el DE y con H y L, que forman el HL.

El Acumulador forma pareja con el registro F, pero solamente para almacenarlos (PUSH AF) o recuperarlos del Stack (POP AF). Este es el único caso donde A y F aparecen juntos.

Un registro doble no es un registro aparte, sino que su contenido es el mismo que el almacenado por los registros simples que lo forman. Si B contiene 00h y C contiene 01 h, el registro BC co1tendrá 0001 h. Como ya hablamos, el registro que almacena el byte más significativo es el que aparece en primer lugar en el nombre del registro doble (B en BC, D en DE y H en H L).

Un ejemplo podría ser LD HL, 4000H (código objeto 210040), que introduce en el registro doble HL el valor 4000H. Esto equivaldría a dos instrucciones de direccionamiento inmediato que afectarán a los registros H y L.

El código de la instrucción está formado por tres bytes. Lógicamente, el primero indica la operación de carga de valores, y los otros dos son el dato a introducir. El valor máximo que permite almacenar dos bytes es el FFFFH o 65535 d. Este valor se divide en dos partes, un byte alto o más significativo, y un byte bajo o menos significativo.

No me cansaré de repetir que el código objeto de este tipo de instrucciones y las que usan dos bytes para expresar un dato presentan una particularidad. Observa que el dato que se debe introducir en el byte bajo o menos significativo aparece en primer lugar. En el ejemplo es el valor 00h, que corresponde al contenido de L. Tras él se encuentra el byte más significativo que será el contenido del registro alto, el valor 40H. Juntos expresan el número 4000H / 16384d.

Si queremos introducir un dato en un registro doble, primero hemos de encontrar su equivalente hexadecimal (para ello puedes usar el programa DEC-HEX de este libro). Después es necesario encontrar el código, cuyo significado exprese la carga de un dato en un registro doble. Estos códigos puedes encontrarlos en las tablas del Apéndice del libro. No debemos olvidar cambiar de orden el byte alto por el byte bajo.

214000	=	LD HL, 0040H	=	Cargar 64	decirnal en HL
210040	=	LD HL, 4000H	=	Cargar 16384	decimal en HL

Esta es una de las reglas de oro de la programación en código máquina con el Z80 que nunca debes olvidar:

"EL BYTE MENOS SIGNIFICATIVO SE ALMACENA SIEMPRE
EN PRIMER LUGAR"

Este tipo de introducción de información se denomina **direccionamiento inmediato extendido**. En realidad es una éxtensión del método anterior, pues con tres bytes se expresa lo mismo que con cuatro en direccionamiento inmediato.

La transmisión de información de un registro a otro se realiza de un modo muy sencillo. Para ello necesitamos solamente de un byte de información. Imagina que el registro H contiene el valor 40H, y el registro A contiene el valor D7H.

La instrucción LD A, H con código objeto 7CH traspasaría el contenido de H al registro A. Tras esta operación, el Acumulador contiene 40H. Su valor anterior D7H no existe ahora está borrado. El registro H sigue manteniendo su contenido, 40H. No ha sido afectado por la instrucción.

Los registros de uso general pueden combinarse, tanto entre sí como con el Acumulador.

Este tipo de direccionamiento se denomina **por registro**. El direccionamiento por registro es solamente lícito para registros simples, nunca dobles. La instrucción

LD BC,HL

no existe. Sería necesario simularla con dos órdenes direccionamiento por registro:

LD B,H
LD C,L

El usuario puede trabajar con el contenido de las direcciones de memoria, tal y como lo hacen las órdenes PEEK y POKE.

En código máquina se utilizan para ello unos códigos objeto determinados. En lenguaje ensamblador aparecen los mnemónicos acompañados de unos paréntesis.

LD HL, (4000H) Codigo Objeto ED680040

Los paréntesis son un símbolo que indican "Contenido de". La instrucción introduce en el registro bajo, el registro L, el contenido de la dirección expresada entre paréntesis, 4000H. El registro alto, el registro H, es cargado con el contenido de la dirección de memoria expresada entre paréntesis más uno, 4001 H. El contenido anterior de HL, cualquiera que sea, sería borrado.
El equivalente BASIC sería:

LET L = PEEK 16384
LET H = PEEK (16384+1)

puesto que 4000H es igual a 16384. Cualquiera de las direcciones de memoria del aparato están a nuestro alcance, pues siempre se moverán en el rango 000H a FFFFH/0 a 65535 d.

La ausencia de paréntesis provocaría un significado completamente diferente. LD HL, 4000H carga el registro doble con el valor 4000H, mientras que LD HL (4000H) carga el mismo registro con el contenido de aquella y la siguiente dirección de memoria.

Esta vez son dos bytes los que indican la operación a realizar. Los dos primeros bytes son dos códigos de operación, y los otros dos, con el orden byte alto-byte bajo invertido, forman una dirección de memoria, que es utilizada por los códigos de operación.

Debe resaltarse que por ser el registro HL el registro doble más utilizado por el microprocesador, existen dos códigos objeto diferentes con este mismo significado. Uno ya lo conocemos, y tiene la longitud de 4 bytes. El otro ofrece la ventaja al programador de tener 3 bytes de longitud y ocupar menos memoria. LD HL (4000H) puede expresarse con:

ED680040
ó
2A0040

Todos los registros dobles, excepto el AF que aunque se empareje en unas instrucciones, no actúa como un registro doble convencional, pueden hacer uso de este tipo de direccionamiento. Para ello se usan códigos objeto de 4 bytes.

Incluso los registros IX, IY, SP y PC son accesibles. Los tres primeros tienen un código objeto determinado. El registro PC se modifica con las instrucciones de salto absoluto (JP NN).

Lo importante en este tipo de instrucciones es la aparición de una dirección de memoria, y que el microprocesador la interprete como tal. Este método de manejo de datos se denomina **direccionamiento extendido.**

El direccionamiento extendido no se restringe solamente a introducir en los registros el contenido de direcciones de memoria. También puede realizar lo contrario: introducir en direcciones de memoria el contenido de los registros.

LD (4000H) , HL   Código Objeto   ED630040

La dirección de memoria aparece en primer lugar, y se escribe entre paréntesis. El significado de esta instrucción es el de introducir corllo co.ntenido de la dirección de memoria 4000H, el valor que tenga el registro bajo, el registro L. El contenido de la dirección de memoria 4001 H será igual al valor que tenga el registro alto, el registro H. Lo contenido anteriormente por ambas direcciones será borrado. El equivalente BASIC nos daría la orden POKE:

```
POKE 16384, HL - INT ( HL/256) *256
POKE 16385, INT (HL/256)
```

Lo expresado tras la coma en estos comandos BASIC forma uno de los métodos para averiguar los equivalentes decimales del byte alto y bajo de un número decimal. Si HL es igual a 32767 d, L debe contener 32767-INT (32767/256) * 256 = 255 d = FFH, y H debe contener INT (32767/256) = 127 d = 7FH. Esto es cierto al comprobar que 32767d = 7FFFH.

La estructura del código objeto es similar a la del anterior. Los dos primeros bytes son dos códigos de operación, y los dos últimos forman una dirección de memoria, cuyo byte alto y bajo están invertidos.

Por ser el registro HL el más usado, existe un código objeto determinado que equivale a este de cuatro bytes. Nos encontramos en el mismo caso que al introducir en un registro el contenido de una dirección de memoria.

LD (4000), HL puede ser expresado con

ED630040
ó
220040

con la ventaja de que el segundo ocupa un byte menos en memoria.

Podemos introducir en cualquier dirección de memoria el contenido que almacene cualquier registro, excepto el AF y el PC. De este modo, también los registros BC, DE, IX, IY y SP pueden ser utilizados. El registro PC no cuenta con una instrucción como

JP (4000H)

Para simularla, habría que seguir los siguientes pasos:

LD HL,4000H
JP (HL)

pero esta última instrucción no corresponde al modo de direccionamiento extendido, que es el que tratamos ahora.

Todas estas modalidades de direccionamiento extendido que hemos visto, utilizaban registros dobles, y, bien "escribían" en dos direcciones de memoria a la vez, bien las "leían".

Solamente existe un registro simple con el cual podemos operar en direccionamiento extendido, el Acumulador.

Para "leer" el contenido de una dirección de memoria determinada, por ejemplo 4000H, se usará la instrucción :

LD A, (4000H)   Código Objeto   3A0040

Es una instrucción de 3 bytes. El primero determina la operación y los otros dos una dirección de memoria, con el orden byte alto-bajo invertido.

Por otra parte, para introducir información en una dirección de memoria, se usará la instrucción:

LD (4000H),A   Código Objeto   32004

Es con esta y otras muchas instrucciones más con las que se advierte la potencia e importancia que puede llegar a tener el Acumulador .

Una vez se haya entendido el direccionamiento extendido, es muy sencillo comprender otro sistema para acceder a datos que se basa en aquél. En el direccionamiento extendido aparece una dirección de memoria como parte inseparable de la instrucción. Esta se define por medio de 2

bytes, con su orden byte alto-bajo invertido. En este nuevo método, la dirección de memoria se define por medio de un registro doble. Recibe el nombre de **direccionamiento indirecto por registro**.

LD A, (HL)   Código Objeto   7E

En este ejemplo, el Acumulador se carga con el contenido de la dirección de memoria que expresa el registro doble HL. Si éste fuera igual a 4000H, la instrucción sería equivalente a

LD A, (4000H)

El direccionamiento indirecto por registro suple la dirección de memoria por un registro doble, normalmente HL. La instrucción tiene la longitud de un byte. En el mnemónico aparece el registro doble entre paréntesis, que aportan el significado "Contenido de". No solamente podemos utilizar el Acumulador para cargar datos en él. El resto de los registros simples de uso general están a nuestra disposición:

LD B, (HL)   /   LD C, (HL)   /   LD D, (HL)   etc

El registro doble que debe suplir la dirección de memoria es normalmente HL, el registro doble preferido por el microprocesador. Pero si usamos el Acumulador para cargar datos en él, se pueden utilizar los registros dobles BC y DE:

LD A, (BC)   Código Objeto   0A
LD A, (DE)   Código Objeto   1A

Lamentablemente no está permitida una instrucción del formato:

LD B, (BC)   ó   LD E, (BC)   ó   LD D, (BC)

Tampoco existe una instrucción que nos cargue en un registro doble el contenido de dos direcciones de memoria expresadas en un registro doble. De esta manera, una instrucción como

LD DE, (HL)

no existe. Para simular esto con direccionamiento indirecto por registro, hay que realizar dos pasos:

```
LD HL,4000H
LD E, (HL)
INC HL
LD D, (HL)
```

Lo que acabamos de hacer introduce en un registro simple el contenido de una dirección de memoria. Pero también es posible realizar lo contrario: introducir el contenido de un registro en una dirección de memoria expresada por un registro doble:

LD (HL),A   Código Objeto   77

Esta instrucción introduce el valor que ese momento tenga el acumulador en la dirección de memoria expresada por el registro doble HL. Si HL fuera igual a 4000H, equivaldría a LD (4000H), A. El registro doble que se utiliza para expresar la dirección de memoria es normalmente HL. Como anteriormente, si se usa el Acumulador como el registro que contiene el valor, BC y DE pueden ser usados:

LD (BC),A   Código Objeto   02
LD (DE),A   Código Objeto   12

Otro registro que no sea el Acumulador no puede ser utilizado conjuntamente con un registro que no sea el HL. Una instrucción como

LD (BC) ,C   ó   LD (DE),L   ó   LD (DE),D

no existe.

El direccionamiento indirecto por registro permite introducir una constante en la dirección de memoria expresada por el registro HL. Así

LD (HL) , FFH   Código Objeto   36FF

introduce el valor FFH /255 d en 4000H, si HL fuera igual a esa dirección. Observa que el código objeto de esta instrucción es de dos bytes de longitud. El primero es el código de operación, y el segundo el byte de datos. Modificando este último, se introduciría otro valor en la dirección de memoria expresada mediante HL.

No se puede trabajar con los registros que no son de uso general, si exceptuamos el Acumulador y el registro PC. El primero de ambos ha sido ya analizado. El registro Contador de Programa se modifica con direccionamiento indirectc por registro por medio de

JP (HL)   Código Objeto   E9

El control del programa salta a la dirección de memoria expresada por el contenido de las direcciones de memoria HL y HL + 1. La primera formará el byte bajo, y la segunda el byte más significativo.

Ahora debes practicar con un ejercicio: una pequeña rutina en código máquina que introduzca en todas las posiciones de la memoria de pantalla el valor 255. Para ello tienes que usar direccionamiento indirecto por registro, e instrucciones que ya hemos estudiado. (Pista: usa un registro doble como contador, y comprueba si ha llegado a su fin con el indicador de cero).

Otro método de direccionamiento parecido a los dos anteriores es el que utiliza los registros índice IX e IY. Se denomina **direccionamiento indexado** y tiene el siguiente principio.

Los registros IX e IY almacenan direcciones de memoria, tal y como podrían hacer BC, DE o HL. El contenido de esas direcciones de memoria puede ser recogido o modificado, tal y como ocurría con el direccionamiento extendido o indirecto por registro.

La diferencia estriba en que la propia instrucción permite modificar el valor de IX o IY. Al registro índice se le puede sumar un número en complemento a dos (negativo 0 positivo). De esta manera podemos trabajar también con las direcciones de memoria en el rango (IX-128) a (IX + + 127) o (IY-128) a (IY+ 127).

La instrucción adopta la siguiente forma:

LD A, (IX + desplazamiento)   Código Objeto   DD7EXX

A partir de ahora utilizaré en los ejemplos solamente el registro índice IX, que puede ser reemplazado siempre por IY. En este caso, el código objeto será diferente.

La instrucción consta de tres bytes. Los dos primeros son códigos de operación. Es el último el que indica el desplazamiento. Si IX es igual a 4000H, la instrucción

LD A, (IX + 80H)   Código Objeto   DD7E80

introduce en el Acumulador el contenido de la dirección de memoria (IX - 128) o 3F80H, pues 80H es el hexadecimal correspondiente al complemento a dos -128.

Si quisiéramos introducir en el registro A el contenido de IX, sin sumarle ningún desplazamiento, la instrucción sería

LD A, (IX + 00H)   Código Objeto   DD7E00

El límite superior lo encontraríamos al sumar al registro Indice el valor 7FH, que corresponde al complemento a dos + 127.

También están a nuestra disposición los registros de uso general. Por supuesto, al utilizar otro registro, la instrucción y el código objeto experimentan una modificación.

Lamentablemente no es posible utilizar en este modo de direccionamiento los registros dobles. De esta manera, una instrucción como

LD HL, (IX + 0AH)

no existe.

Queda solamente analizar el proceso inverso al que acabamos de ver. También mediante direccionamiento indexado es posible modificar el contenido de una dirección de memoria. Una instrucción de este tipo tendría la forma

LD (IX + desplazamiento),A   Código Objeto   DD77XX

Los dos primeros bytes son códigos de operación. El último indica el desplazamiento que debe sufrir el registro índice. Al igual que anteriormente, nos movemos en el rango -128 a + 127 direcciones de memoria. Imagina que A vale 255 y que IX vale 4000H. La instrucción

LD (IX + 80H),A   Código Objeto   DD7780

introduce en la dirección de memoria (IX + 80H) que es igual a 3F80H el valor que contiene A, 255. Si queremos afectar a otras direcciones de memoria no hay más que modificar el byte de desplazamiento. Los límites serán los mismos que antes: -128 a + 127.

Cualquier registro simple de uso general más el Acumulador puede hacer uso del direccionamiento indexado, que permite usar una constante n en el rango 0 a 255 como valor a introducir .

El mnemónico es el siguiente:

LD (IX + desplazamiento), constante Código Objeto DD36XXXX

La instrucción consta de cuatro bytes. Los dos primeros bytes son códigos de operación. El tercer byte se interpreta como desplazamiento del registro índice, y el cuarto como constante. La instrucción para introducir el valor 255 en la dirección (IX + 127) tendría el siguiente formato:

LD (IX + 7F) , FF   Código Objeto   DD367FFF

El registro PC puede ser afectado por los registros IX o IY.

JP (IX) modifica el registro PC al contenido de las direcciones de memoria (IX) e (IX + 1), tal y como lo hace JP (HL).

El verdadero interés de estos registros índice se basa en el establecimiento de tablas de 255 elementos como máximo a los que se puede acceder con el direccionamiento indexado. La técnica y el procedimiento para estas tablas pertenece ya a un nivel superior de programación.

El **direccionamiento relativo** se basa en la modificación del registro PC por medio de instrucciones JR. A su contenido actual se le añade un número en complemento a dos. De esta manera se accede al Contador de Programa, modificándolo en el rango -128 a + 127.

JR F8   significa PC = PC - 8

La suma del acumulador y un registro o del acumulador y una constante constituye también un método para acceder a los registros y datos. Esta instrucción y otras que afectan solamente a un registro en concreto, el acumulador, pertenecen a un modo de direccionamiento denominado **direccionamiento implícito**. El ejemplo de la suma o la resta fue explicado con extensión en el capítulo cuarto.

El microprocesador ha preparado una instrucción especial que con un solo byte de información iguala a un CALL. Se utiliza para subrutinas muy usadas, ubicadas en las primeras direcciones de la memoria. El mnemónico recibe el nombre de RST (de ReStaT), y las subrutinas se encuentran en las direcciones 0000, 0008, 0010, 0018, 0020, 0028, 0030 y 0038. Como podrás observar, el byte más significativo de estas direcciones es siempre igual a cero. Por esto, el método para acceder a los datos se denomina **direccionamiento por página cero**.

RST 10H Código Objeto D7

Existe todavía un último caso en que una instrucción puede afectar a un dato o un registro. Se puede modificar el estado de cada uno de los ocho bits del byte de un registro o del byte de una dirección de memoria. El modo recibe el nombre de **direccionamiento de Bit**. El tema será tratado con extensión en el capítulo noveno.

En referencia al efecto de estas instrucciones sobre los indicadores, hay que decir que los primeros siete modos de direccionamiento (del inmediato al relativo) no afectan al indicador de arrastre (Carry Flag).

El estado que puedan adoptar el resto de los indicadores es muy diverso e imposible de resumir.

8
Instrucciones Lógicas

El objetivo de este apartado es mostrar las tres instrucciones lógicas incluidas en el microprocesador Z80 y algunos de sus usos.

El término "lógica" difiere en este contexto del significado que adopta normalmente. Esta "lógica" se basa en el Algebra de Boole, científico inglés del siglo pasado que estructuró la matemática de los sistemas lógicos. Para ello ideó una serie de símbolos (símbolos booleanos) y de instrucciones (instrucciones booleanas). Estas últimas han sido incluidas en la gama de instrucciones de los microprocesadores y serán analizadas a continuación.

1. AND

No debemos confundir este AND con el que aparece en sentencias BASIC en la firma IF X = 1 AND Y = 0..., aunque sí tiene algo en común con ella.

Muchas de las instrucciones de un ordenador se comprenden mucho mejor si se expone un ejemplo análogo de nuestra vida, un ejemplo más próximo a nosotros.

Imagina una tubería con dos llaves en dos puntos diferentes. El agua fluirá solamente si la primera llave y la segunda están abiertas. Observa la importancia de la palabra y (AND en inglés). Si una u otra están cerradas, el agua no puede fluir. Acabamos de definir la operación lógica AND.

Podemos sustituir los estados abierto / cerrado de las llaves y del estado agua fluye /agua no fluye por símbolos lógicos.

LLave cerrada = 0	Agua no f luye = 0
LLave abierta = 1	Agua fluye = 1

Estos símbolos representan los dos estados lógicos que, en este caso, la llave y el agua que fluye, pueden adoptar. Cada llave puede adoptar dos estados, la combinación de dos llaves, cuatro.

Cada una de estas combinaciones tiene como resultado otro estado lógico (Agua fluye x.1 ; Agua no fluye = 0). El conjunto puede ser resumido en una tabla:

		Llave 2	
		0	1
Llave 1	0	0	0
	1	0	1

Verifica tú mismo todas las posibles combinaciones de las llaves (0 y 1) con los resultados correspondientes del agua (0 y 1).

El estado lógico 0 y 1 se utiliza igualmente en otros contextos. Por ejemplo y como vimos al principio, la ausencia o presencia de tensión se pod ía interpretar con un 0 y un 1. Esta ausencia o presencia de tensión tiene la significación de un mensaje. El 0 y el 1 también lo representan. Es de aquí de donde se deriva el bit como unidad mínima de información, de un mensaje. El bit se simboliza con los estados lógicos 0 y 1.

La instrucción lógica AND puede ser utilizada con los diferentes estados de dos bits. Las dos combinaciones posibles de cada uno de ellos (cuatro en total) nos darán cuatro resultados.

		Bit 2	
		0	1
Bit 1	0	0	0
	1	0	1

La operación lógica AND da como resultado un 1, si ambos estados lógicos ANDeados son igual a 1.

El siguiente paso es la instrucción mnemónica AND. Esta es la que se utilizará en programas assembler. La base de la instrucción es la misma que la de la operación.

AND necesita de dos elementos para comparar sus estados y extraer de ellos un resultado. En la instrucción, uno de ellos es cada uno de los bits del Acumulador, y el otro es cada uno de los bits .del contenido de un registro, una constante 0 una dirección de memoria, dependiendo del modo de direccionamiento. El mnemónico AND opera no con un bit, sino con cada uno de los 8 que forman el byte. Para hacernos una idea más clara debemos utilizar la base dos a la hora de realizar la operación.

Acumulador	0 0 1 1 1 0 1 1
Constante 170d	1 0 1 0 1 0 1 0

Resultado AND 170d	0 0 1 0 1 0 1 0

En este ejemplo el acumulador contiene el valor 00111011 b /59 d / 3BH. Se procesa la instrucción AND 10101010b /170 d / AAH.

Se siguen los siguientes pasos:

- El contenido del acumulador y la constante se pasan a base dos.

- La operación lógica AND se realiza con cada uno de los bits del Acumulador y el bit con el número de orden correspondiente de la constante. Es decir, se ANDea el bit 0 del registro A con el bit 0 de la constante; el bit 1 del Acumulador con el bit 1 de la constante, yasí hasta el bit 7 del Acumulador, que se ANDea con el bit 7 de la constante. Dependiendo del estado de cada uno de los bits, existirá una u otra solución. En el ejemplo la solución fue 00101010 b /42 d /2AH.

- Muy importante es saber que este resultado va a ser el nuevo contenido del registro A.

Uno de los dos elementos debe ser, inexorablemente, el Acumulador. El otro puede ser una constante, un registro o el contenido de una dirección de memoria.

En el ejemplo utilizamos el modo de direccionamiento inmediato, por tanto una constante. El dato forma parte de la instrucción. El código de operación es en este caso E6H. El código objeto completo de la instrucción AND 170d/AAh será el E6AA.

Otro modo de direccionamiento implicaría el uso de otro código de operación. Si se realiza la operación lógica con un registro, que debe ser simple, la longitud de la instrucción será sólo de un byte.

AND D realiza la operación lógica AND con el contenido del Acumulador y el contenido del registro D. El resultado será introducido en el registro A, borrando el anterior. Por otra parte, el contenido del registro D permanece invariable tras la operación.

El modo de direccionamiento indirecto por registro (AND (HL)) e indexado (AND (IX + desplazamiento)) pueden ser igualmente usados.

El efecto de la instrucción AND en los indicadores es el siguiente:

- El indicador de arrastre toma siempre el valor 0 tras la ejecución de una instrucción AND, independientemente del resultado que se derive de la operación.

- El indicador de cero tomará el valor 1, si el resultado de la operación ha sido igual a 0, es decir, que no han existido nunca dos bits con el mismo número de orden y con valor 1. En cualquier otro caso, este indicador toma el valor 0.

2. OR

Tampoco esta instrucción es la misma que encontramos en ciertas sentencias BASIC IF X = 1 OR Y = 0....

Imagina esta vez dos tuberías paralelas que se unen en un punto determinado. Cada una de ellas tiene, antes de llegar al punto de unión, una llave que permite o impide el paso del agua.

El líquido llegará al punto de unión si la primera o la segunda llave están abiertas. Sólo en caso de que las dos permanezcan cerradas, el agua no fluirá.

En este caso, es también bastante práctico el sustituir los estados abierto/cerrado de las llaves y agua fluye / no fluye por los símbolos lógicos 0 y 1 :

		Llave 2	
		0	1
Llave 1	0	0	1
	1	1	1

El líquido fluye (estado 1) si una de las dos llaves está abierta (estado 1) . Esta es la operación OR (en castellano, O).

Verifica aquí también todos los resultados derivados de las dos posiciones lógicas de la llave (abierto/cerrado) .

El estado lógico 0 y 1 se utiliza igualmente al determinar la unidad de información, el bit.

Tal y como hicimos anteriormente, podemos reemplazar los elementos "llave" y "agua" por los bits y el resultado de ambos:

		Bit 2	
		0	1
Bit 1	0	0	1
	1	1	1

De cuatro resultados posibles, tres de ellos dan un 1. Sólo en el caso que ambos bits sean igual a 0, la solución será cero.

El mnemónico OR se basa en esta última operación lógica. Por una parte se encuentran, al igual que con la instrucción AND los 8 bits del acumulador. Como elemento a ORear se puede encontrar una constante, un registro simple o el contenido de una dirección de memoria.

Imagina que el acumulador contiene el valor 59d / 3Bh. Una operación OR se realiza con la constante 170d/ AAh.

Para que podamos comprender el resultado, ambos elementos se pasan a sistema binario. La operación lógica OR se realiza con cada uno de los bits del mismo orden. El resultado final pasa a ser el nuevo contenido del Acumulador .

```
Acurnulador            0 0 1 1 1 0 1 1

Constante 170d         1 0 1 0 1 0 1 0
                         ------------------
Resultado AND 170d   1 0 1 1 1 0 1 0
```

En este ejemplo, la solución es igual a 186d /BA H. Verifica el resultado obtenido de cada uno de los bits.

El código objeto de la instrucción OR 170d consta de dos bytes. El primero es el código de operación, y el segundo el dato.

Si se usara otro modo de direccionamiento (por registro, indirecto por registro) la longitud de la instrucción es menor, sólo de un byte.

Al igual que ocurría con la instrucción AND, el contenido del registro o de la dirección de memoria, en caso de utilizarlos con OR, no efectúan ninguna modificación.

Es importante saber siempre que el resultado de la operación lógica será introducida en el Acumulador.

La instrucción OR influye de esta manera sobre los indicadores:

El indicador de arrastre toma tras la ejecución de la operación lógica OR el valor 0.

El indicador de cero tomará el valor 1 en caso que el resultado de la operación sea igual a 0. Esto ocurre solamente si el Acumulador es igual a 0, o si el byte con el que se va a realizar la operación OR es igual a 0. En cualquier otro caso, el indicador de cero toma el valor 0.

3. XOR

No existe ninguna función BASIC que siquiera se asemeje en el nombre a esta operación lógica. El nombre es una abreviatura para Exclusive OR (O-Exclusivo).

Hay que buscar ejemplos análogos de la vida para comprender el significado que aporta XOR.

Imagina un ascensor con solamente dos botones, el de subida y el de bajada. Pulsando el de subida, el ascensor obedece y efectúa una subida. Asímismo, si se pulsa el botón de bajada, el ascensor obedece y baja. En el caso de que ningún botón sea pulsado, el aparato no se mueve. Pero si ambos botones se accionan a la vez, el ascensor no sabrá a cual obedecer, y permanecerá quieto.

Los símbolos lógicos 0 y 1 pueden ser utilizados para representar los estados de botón pulsado / botón no pulsado y de ascensor obedece / ascensor no obedece.

Botón no pulsado = 0	Ascensor quieto = 0
Botón pulsado = 1	Ascensor en marcha = 1

y con estos símbolos, se podría elaborar una tabla:

		Botón n.2	
		0	1
Botón n.1	0	0	1
	1	1	0

Esto sería el resultado de una operación XOR. Las combinaciones de los cuatro elementos (dos ceros y dos unos) entre sí dan como resultado 4 estados. Dos de ellos son ceros, y otros dos son unos. El resultado será cero, si ambos elementos XOReados son iguales. La solución 1 aparece si los dos elementos son diferentes.

Al igual que hemos hecho anteriormente con AND y OR, podemos interpretar los diferentes estados de dos bits y aplicarles la operación XOR. Con ello podemos crear una tabla de resultados:

		Bit n.2	
		0	1
Bit n.1	0	0	1
	1	1	0

Las cuatro posibles soluciones coinciden con las obtenidas en el ejemplo del ascensor. Sólo si ambos bit son de diferente estado es igual a uno.

Ahora sólo queda explicar la instrucción mnemónica XOR. Se basa en lo mismo que la instrucción booleana XOR, que esla que acabamos de ver. Se diferencia de ella en que son 0 los bits o

los estados lógicos con los que se opera. Estos se encuentran contenidos en el Acumulador. Los otros 8 bits, que servirán para ejecutar la orden XOR, provienen de una constante, del contenido de un registro simple o del contenido de una dirección de memoria. Veamos como operaría XOR con una constante:

Acurnulador	0 0 1 1 1 0 1 1
Constante 170d	1 0 1 0 1 0 1 0

Resultado AND 170d	1 0 0 1 0 0 0 1

En el ejemplo, el Acumulador contiene el valor 00111011 b / 59d / 3Bh. La constante es igual a 10101010b / 170d / AAh. El cambio de los números a base binaria nos permite poder comprobar el resultado. En este caso, éste es igual a 145d / 91 H. Cada bit del registro A se XORea con su bit correspondiente de la constante. El resultado pasa a ser introducido en el Acumulador.

Observa que la solución solamente es cero, cuando acumulador y byte utilizado son iguales.

Otros modos de direccionamiento pueden ser utilizados:

Por registro : XOR B, XOR C...
Por registro indirecto : XOR (HL)
Indexado : XOR (IX + desp), XOR (IY + desp)

El efecto de la instrucción XOR sobre los indicadores de arrastre y de cero es el siguiente:

- El indicador de arrastre toma el valor 0 tras la ejecución de la instrucción XOR.

- El indicador de cero tomará el valor 1 si el resultado obtenido de la operación, resultado que se introducirá en el registro A, es igual a 0. En caso contrario, resultado diferente de cero, el indicador de cero tomará el valor 0.

4. USOS DE LAS INSTRUCCIONES LOGICAS

Comprender el funcionamiento de las instrucciones lógicas es sencillo. Más complicado resulta comprender para qué sirven y cómo se utilizan. Este apartado muestra ciertos usos que pueden adoptar.

- La instrucción AND puede ser usada para "enmascarar" ciertos bits. Esto significa poner a cero ciertos bits de un byte. Observa el siguiente ejemplo:

AND 7F opera con el contenido del Acumulador y la constante 7FH. Fíjate en la forma binaria de la constante 7 Fh :

7Fh = 0 1 1 1 1 1 1 1 b

No se puede saber con exactitud el resultado, pero de lo que se puede estar seguro, es que el 7 bit de la solución va a ser igual a cero. La razón para ello estriba en que el 7 bit de 7Fh es cero y para que AND de un 1 como resultado, tiene que haber 2 bits con valor 1. El resultado será menor a 80h, menor a 128 d.

El siguiente programa BASIC llena la memoria de atributos con la función RND:

```
10 FOR F = 22528 TO 23295
20 POKE  F,RND*256
30 NEXT F
```

Ahora debes hacer tú una rutina en código máquina que elimine la función FLASH 1 de toda la memoria de pantalla. Por supuesto deberás utilizar el mnemónico AND.

Observa el efecto de la instrucción AND A. Es totalmente indiferente el contenido del Acumulador para el resultado. El resultado obtenido de AND A es siempre el contenido previo del Acumulador:

Acurnulador	0 1 0 1 0 1 0 1
AND A	0 1 0 1 0 1 0 1
	-------------------------------
	0 1 0 1 0 1 0 1

La ventaja de esta instrucción es su efecto en los indicadores. El de arrastre se pone a cero, sin modificar el contenido de ningún registro, ni siquiera el del Acumulador.

Poner a cero el indicador de arrastre puede ser interesante cuando se realizan operaciones aritméticas que impliquen su uso y no se conoce en ese momento (ADC, SBC).

Con la instrucción AND podemos comprobar igualmente el estado de cada uno de los bits del Acumu lador . Podemos saber, por ejemplo si el 5 bit está puesto a .uno mediante la instrucción AND 00100000b / 32d / 20h. Si es así, el indicador de cero tomará el valor 1. Una instrucción de salto condicional (JP Z, JR Z) nos enviará a otra parte del programa.

El mnemónico OR puede ser usado para poner a 1 a ciertos bits de un byte. Es la función inversa a AND. Veamos que puede ocurrir con OR 80h / 10000000 b. El 7 bit del byte es el único que está a uno. El 7 bit del resultado tiene que tener este mismo bit con el estado igual a 1 , dado la acción de la operación OR.

Esto equivale a sumar al acumulador 128d, sin provocar un sobrepasamiento del arrastre. Las consecuencias de OR sobre los indicadores pueden ser aprovechadas. Analicemos el siguiente listado assembler .

```
=======================================================================
Especificaciones : "Prog 8.1" par"a ZX Spectrum 16K/48K.
Descripción General : Uso de OR con el indicador de cero. Muestra
 un bucle.
Longitud : 9 bytes .
Entrada : Ningún requerimiento.
Salida : Ningún requerimiento.
Registros Usados : A,D y E.

7000        00100          ORG 7000H       ;
7000 11FFFF 00110                LD DE,FFFFH    ;
7003 7A     00120          BUCLE LD A,D         ; ES D = E = 0 ?
7004 B3     00130                OR E           ;
7005 1B     00140                DEC DE         ; DE = DE - 1
7006 20FB   00150                JR NZ,BUCLE    ; SI DE <> 0 ---> BUCLE
7008 C9     00160                RET            ;
0000        00170          END             ;

7003        BUCLE

=======================================================================
```

El programa es en sí un bucle realizado con la instrucción OR y el indicador de cero. El registro DE es utilizado como contador, cuántas veces ha de realizarse el bucle. El valor inicial del registro "contador" es el FFFFH / 65535 d. Las líneas assembler 00120 y 00130 tienen la función de realizar la función OR entre el valor de D y el de E. El resultado de esta operación pondrá a cero el indicador de cero, si ambos son iguales a cero. Ese caso querrá decir que se ha alcanzado el fin del contador.

Observa que la instrucción DEC DE no afecta al indicador de cero, en realidad no afecta a ningún indicador. DEC registro simple sí lo hace, pero cuando se trata de registros dobles, la efectividad de DEC sobre los indicadores se pierde.

La siguiente instrucción repite de nuevo el bucle si el indicador de cero está a cero, lo que quiere decir que todavía no se ha acabado el bucle.

Este sistema puede ser usado como pausa en un programa en código máquina, aunque ésta no sea la verdadera función de OR. Este es el listado BASIC correspondiente al programa 8.1 :

```
10 REM PROG 8.1
20 CLEAR 28650: LET T=0
30 FOR F=28672 TO 28680
```

```
40 READ A: POKE F,A
50 LET T=T+A: NEXT F
60 IF T< > 1339 THEN PRINT "ERROR EN DATA" : STOP
70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
   MAQUINA": PAUSE 0
80 RANDOMIZE USR 28672
90 DATA 17 , 255 , 255 , 122 , 179 , 27 , 32 , 251 , 20l
```

Por último, el mnemónico XOR. Uno de sus usos más corrientes permite poner a cero el acumulador y el indicador de arrastre, en una sola instrucción y en un solo byte:

XOR A realiza la operación lógica XOR consigo mismo. Es indiferente el contenido del Acumulador, pues el resultado siempre será igual a cero. Pruébalo tú mismo con los valores que quieras. Esto equivaldría la instrucción de dos bytes LD A, 0.

Como cualquier instrucción lógica, su ejecución pone a cero el indicador de arrastre.

9

Instrucciones de Manipulación de Bit

Este capítulo analiza las tres instrucciones correspondientes al modo de direccionamiento de bit del microprocesador Z80.

Es este modo el que trabaja directamente con las unidades m ínimas de información, los bits. Por una parte, el programador tiene la posibilidad de modificar el estado de un bit determinado. También puede investigar y adivinar su contenido.

Antes de empezar hay que tener muy claro qué significado tiene el bit en estas instrucciones. Te remito a la primera parte del libro si no tienes claro el concepto.

Imagina que el valor 85d. Su equivalente en base dos es el 01010101 b. Estas cifras binarias son la interpretación de los 8 bits que forman la unidad de información, el byte. Este valor puede ser almacenado en un registro o en una dirección de memoria.

Las tres instrucciones de manipulación de bit afectan o analizan un bit determinado del byte. Pasamos a estudiarlas:

1. SET

Uno de los muchos significados de esta palabra, que no es una abreviatura, es literalmente el de "poner, levantar, construir".

La programación assembler ha escogido esta palabra porque su significado "normal" se relaciona fácilmente con el que obtiene como instrucción assembler .

SET afecta a un determinado bit y lo pone a uno, lo "levanta". Presenta la siguiente forma:

SET n.bit, registro ó dirección de memoria

Tras la palabra SET debe aparecer el número de orden del bit que se quiera poner a uno. Separado por una coma se encuentra el modo de llegar al bit. Veamos el primer caso. El bit es parte del byte de un registro. Una instrucción de este tipo podría ser:

SET 7,A Código Objeto CBC7

Consta de dos bytes. Los dos son códigos de operación. El ejemplo muestra la instrucción que se debe usar para poner a uno el bit número 7 del Acumulador. Observemos los cambios que se efectúan si éste contiene el valor 85d.

Acumulador antes de la operación: 0 1 0 1 0 1 0 1
INSTRUCCION SET 7, A
Acumulador después de la operación: 1 1 0 1 0 1 0 1

Recuerda que es el bit séptimo el que se encuentra más a la izquierda del byte. La cuenta comienza a la derecha con el número de orden cero. La instrucción ha modificado el estado del bit más significativo. El valor total del acumulador ha cambiado de 85d a 213d. La diferencia es de + 128 y se debe precisamente al cambio del último bit.

El resto de los registros simples está a nuestra entera disposición: B, C, D, E, H y L. Pero aún existe un caso que está por aclarar.

Registro D antes de la operación: 1 0 1 0 1 0 0 0
INSTRUCCION SET 3, D
Registro D después de la operación: 1 0 1 0 1 0 0 0

En este ejemplo se trabaja con el registro simple D. La instrucción pone a uno el bit número tres del registro. Pero antes de la operación ya contenía este valor. Realmente el contenido previo a la operación es indiferente. Lo importante es que tras ella se ha cumplido el poner a uno el bit. Si antes ya tenía este estado, entonces no será modificado ni el valor de ese ni de ningún bit. Parecerá que no ha existido operación alguna.

La segunda posibilidad de operar con la instrucción SET se basa en usar contenidos de direcciones de memoria, y no de registros. Para ello se utilizan registros dobles: el HL, IX o IY.

El primero de estos tres debe definir una dirección de memoria absoluta. El contenido de HL será interpretado como tal. La instrucción el siguiente formato:

SET n.bit, (HL)

Imagina que HL es igual a 4000H, y que esta dirección de memoria contiene el valor 0011111b/ 31 d / 1Fh. Observa cómo cambia el contenido tras:

HL = 4000H (4000H) = 0 0 0 1 1 1 1 1
SET 7, (HL)
HL = 4000H (4000H) = 1 0 0 1 1 1 1 1

El contenido del registro doble no experimenta ningún cambio. Permanece antes y después de la operación con el valor 4000h. La instrucción SET afecta al contenido de la dirección de memoria expresada por el contenido del registro doble HL.

Lo que se modifica es el valor que contenga la dirección de memoria, en este ejemplo la, 4000H. Su valor pasa de 00011111b / 31d / 1Fh a 10011111 b/ 159 d / 9F h

La instrucción SET conjuntamente con los registros índice adopta esta estructura:

SET n.bit, (IX/IY + desp)

La instrucción está formada por un total de cuatro códigos. Tres son códigos de operación, los dos primeros y el último. Es el tercero el que se interpreta como un número en complemento a dos, el desplazamiento. De esta manera se consigue un campo de acción de -128 a + 127 con respecto a los valores de IX e IY.

La ejecución de este mnemónico no ocasiona ningún tipo de efecto sobre los indicadores. Antes y después de cada instrucción siguen manteniendo el mismo valor.

2. RES

El mnemónico RES es una abreviatura de la palabra inglesa RESET, cuyo significado es exactamente el inverso al de la instrucción anterior. El verbo inglés RESET adopta en este contexto el significado de "bajar, poner a cero".

Es precisamente el estado de un bit el que se baja o se pone a cero. Al igual que la instrucción anterior, RES afecta al estado de un determinado bit o del contenido de una dirección de memoria. Veamos los dos casos posibles:

Imagina que el registro B contiene el valor 01110111 b / 119d / 77 h y que se efectúa la instrucción RES 0,B.

Valor del registro B antes de la operación:	0 1 1 1 0 1 1 1
RES 0,B	
Valor del registro B después de la operación:	0 1 1 1 0 1 1 0

Como puedes comprobar en el estado de los bit, el único que ha sido modificado es el número cero. Su valor ha pasado de valer uno a valer cero. El bit cero ha sido objeto de la instrucción RES.

Como consecuencia de esto, el valor total del registro ha experimentado un cambio. Ahora contiene el número 01110110 b / 118 d / 76 h.

Como en el caso de la operación anterior, si el bit que se quiere poner a cero ya contiene este valor, parecerá que la instrucción no ha tenido ningún efecto, y el registro almacenará el mismo valor.

Si se quiere operar no con registros, sino con los contenidos de direcciones de memoria, entonces la instrucción adopta otra forma:

RES n.bit, (HL)

La operación afecta al contenido de la dirección de memoria expresada por el registro doble HL. Si HL es igual a 4000H, y el contenido de esta dirección es igual a 10001111b /143 d / 8Fh, la instrucción

LD HL = 4000H (4000 H) = 1 0 0 0 1 1 1 1
RES 1, (HL)
LD HL = 4000H (4000 H) = 1 0 0 0 1 1 0 1

pondría a uno el bit número 1 del byte. De esta manera, el contenido de la dirección de memoria 4000h se modificaría a 10001101b / 141 d / 8Dh. Hay que resaltar que el registro doble HL no experimenta ningún cambio, seguiría siendo igual a 4000h. Sólo cambiaría la dirección de memoria que representa.

La diferencia al utilizar los registros índice se basa en la longitud de la instrucción (4 bytes) y el margen -128 a + 127 direcciones de memoria con respecto a los valores de IX o IY.

RES n.bit, (IX/IY + desp)

Tres de los bytes son códigos de operación, curiosamente el primero, el segundo y el cuarto. Como tercer byte se debe introducir un número en complemento a dos que determina la distancia, positiva o negativa, de la dirección de memoria a afectar con respecto a IX ó IY. Un ejemplo podría ser:

RES 4, (IX+FF) Codigo Objeto DDCBFFA6

Esta instrucción introduciría el valor cero en el bit número cuatro de la dirección de memoria (IX-1), puesto que FFh es la representación hexadecimal del complemento a dos -1.

El efecto que la instrucción RES causa sobre los indicadores es totalmente nulo. Su contenido es igual antes y después de la ejecución de RES.

Piensa que existen exactamente 80 instrucciones diferentes de SET y otras 80 para RES.

8 instrucciorles para cada bit y registro = 8 * 7 = 56
8 instrucciones con acceso por registro indirecto (SET n.bit, (HL)) + 8
16 instrucciones corl acceso por. registro indexado (SET n.bit, (IX/IY + desp) + 16 = 80

¿ Recuerdas el programa en código máquina que debías realizar como ejercicio en el capítulo anterior? Se trataba de una rutina que quitara la función FLASH 1 de la memoria de atributos. Funcionaba conjuntamente con un corto programa BASIC que llenaba la pantalla con colores aleatorios.

Intentaba hacer ahora otra rutina con la misma función, pero usando el mnemónico RES. Al ser un programa más avanzado cumple el mismo cometido, pero en menor longitud.

Recuerda que es el 7.bit de cada una de las direcciones de la memoria de atributos el que controla la función de parpadeo.

3. BIT

El título que encabeza este apartado no responde a la necesidad de explicar lo que es un bit y para qué sirve, porque se supone que tienes el concepto bien claro.

El título que encabeza este apartado es sencillamente el nombre de una instrucción, la tercera, de manipulación del bit.

Si las dos primeras modifican el contenido de esta unidad elemental de información, a uno o a cero, ésta comprueba qué valor tiene.

La instrucción BIT realiza un TEST sobre un bit determinado de un byte, ya sea el contenido de un registro simple o el contenido de una dirección de memoria. Las consecuencias de ese test se verán reflejadas en los indicadores, especialmente el indicador de cero. Presenta el siguiente formato:

BIT n.bit, registro ó dirección de memoria

Aprendemos del siguiente ejemplo, en el que el registro A contiene el valor 01100111 b / 103d/67 h.

```
Acumulador      =      0 1 1 0 0 1 1 1   b
             BIT 2,A
Acumulador      =      0 1 1 0 0 1 1 1   b
         Indicador de cero =  0
```

La instrucción BIT 2, A realiza un test sobre el estado del segundo bit del registro A. Tras la operación, el contenido del registro no ha cambiado. El segundo bit de A sigue siendo igual a 1. La consecuencia de esta instrucción se ve en el indicador de cero. La instrucción ha preguntado: ¿Cuál es el estado del segundo bit de A? El microprocesador se ha dirigido al Acumulador y ha visto que el valor de ese bit es diferente de cero. Por ello ha puesto a cero el Indicador de cero. Recuerda que el cero es su forma de decir "NO" y que el indicador de cero muestra siempre

el resultado de la última operación (0 = fue diferente de cero; 1 = fue igual a cero), o como acabamos de aprender, el estado de un bit determinado (0 = bit diferente de cero; 1 = bit igual a cero).

Probemos ahora con el mismo valor para el registro A, pero con la instrucción BIT 3,A:

Acumulador = 0 1 1 0 0 1 1 1 b
BIT 3,A
Acumulador = 0 1 1 0 0 1 1 1 b
Indicador de cero = 1

Observa que en ninguno de los casos se ve afectado el contenido del Acumulador. El microprocesador sólo mira el estado del bit, sin modificarlo. En esta ocasión el bit número tres es igual a 0. Esta instrucción BIT causa un efecto sobre el indicador de cero diferente a la anterior. Al ser este bit igual a cero, el indicador de cero tomará el valor 1. El microprocesador nos pretende informar que "SI", el bit sí es igual a cero. Y por ello pone un uno en el indicador de cero.

Recuerda la reacción del microprocesador al ejecutar la instrucción BIT . Pondrá el indicador de cero a cero (0) si el bit en cuestión es igual a 1 (-> diferente de cero) . Por el contrario, si el bit es igual a 0, el Z80 pone el indicador de cero a uno (1).

No sólo podemos utilizar el acumulador para estos fines. El Z80 permite comprobar el estado de cualquier bit de los registros de uso general más el acumulador (registros A, B, C, D, E, H y L).

Una instrucción con registro simple (BIT 4, D, BIT 7, C...) utiliza dos bytes de memoria. Ambos son dos códigos de operación.

BIT puede trabajar igualmente con el contenido de una dirección de memoria, que puede estar expresada por medio del registro doble HL o de los registros índice IX/IY.

La instrucción adopta en el primer caso la forma:

BIT n.bit, (HL)

El código objeto correspondiente tiene la longitud de dos bytes. Si H L es igual a 4000h, y el contenido de esta dirección de memoria es 00000000 b/ 0 d / 00h, la instrucción BIT 0, (HL) comprueba el estado del bit número cero, el primero comenzando por la derecha:

HL = 4000H (4000 H) = 0 0 0 0 0 0 0 0
BIT 0, (HL)
HL = 4000H (4000 H) = 0 0 0 0 0 0 0 0
Indicador de cero = 1

BIT no ha modificado ni al registro doble HL, ni al contenido de la dirección de memoria que representa. Sólo ha afectado al indicador de cero, que se ha puesto a uno.

Si se quieren usar los registros índice IX e IY con la instrucción BIT hay que conocer su formato:

BIT n.bit, (IX/IY + desp)

La instrucción consta entonces de cuatro bytes. Los dos primeros y el último son códigos de operación, que determinan la misma y el bit a chequear. El tercer byte se interpreta como código de desplazamiento, en el rango -128 a + 127 (80H a 7Fh).

Hay que subrayar que el indicador de arrastre no se ve afectado en ningún momento por la instrucción BIT .

Quizá estés pensando en la similitud de esta instrucción con la instrucción lógica AND. Esta también puede ser utilizada para adivinar el estado de un bit determinado, como vimos en el capítulo anterior. Lo cierto es que ese uso de AND se asemeja mucho a la función de BIT. La diferencia es que AND afecta al contenido del acumulador, mientras que BIT ni siquiera hace uso de él.

En cuestión de ocupación de memoria, la instrucción BIT es más económica en todos los casos excepto al comprobar el estado de un bit del Acumulador. En este caso, ambas instrucciones ocupan el mismo número de bytes.

```
LD A, (4000H)    =    LD A, (4000H)
AND 32D          =    BIT 5,A
```

Pero :

```
LD B,A           =    BIT 5,B
AND 32D
```

ó

```
LD A,(IX+02)     =    BIT (IX+02)
AND 32D
```

En estos últimos casos, el ahorro de memoria es de un byte.

Para concluir no queda más que decir que los mnemónicos SET, RES y BIT no se utilizan muya menudo, pero cuando llega este momento, sirven de mucha ayuda.

Parte tercera

PROGRAMAS

10

¡Usa el Código Máquina!

Nos encontramos ahora en un punto muy importante del aprendizaje de un lenguaje, ya sea BASIC, FORTH, COBOL o en nuestro caso, Código máquina. Se caracteriza por el conocimiento de las formas de comunicación, de las instrucciones y por no saber **cómo utilizarlas** en un programa.

Piensa en el pasado y recuerda cuando adquiriste tu ZX Spectrum y empezabas a conocer el lenguaje BASIC. En realidad los comandos son muy sencillos de comprender, pero al intentar usarlos, al intentar realizar uno mismo los programas sobrevienen los problemas. Para llegar a saber programar bien en BASIC tuviste necesidad de analizar muchos, comprender lo que hacen y por qué lo hacen.

Vuelve de nuevo al presente y reflexiona. En este momento te encuentras en la misma situación. Conoces muchas de las instrucciones del lenguaje ensamblador. Sabes moverte con términos como bit, byte, dirección de memoria, registro, indicador, pero si intentas escribir tú mismo un programa, comienzan las dificultades.

Esta tercera y última parte del libro tiene la función de mostrarte, en varios capítulos, programas y rutinas en código máquina. Aparte de que te van a ser útiles, conocerás cómo están hechos y por qué hacen lo que hacen.

Comenzaremos por lo más sencillo, una serie de rutinas ROM, para finalizar con un programa relativamente largo (más de 500 bytes) que imprime en pantalla cualquier variable contenida en un programa BASIC.

RUTINAS DE LA MEMORIA "ROM"

La memoria de lectura está formada en gran parte por rutinas o subrutinas, escritas en código máquina. Estas son utilizadas por el sistema operativo. El traductor traduce, y valga la redundan-

cia, una sentencia BASIC a código máquina. El sistema operativo utiliza entonces las rutinas y subrutinas correspondientes de la memoria ROM que ejecutan el comando BASIC.

El programador puede utilizar ciertas de esas rutinas 0 subrutinas de la ROM para sus propios programas. Lo que debe conocer entonces son las direcciones de memoria donde esas rutinas están ubicadas y la forma de accederlas. Con esto último me refiero a qué registros deben contener qué valores para que la rutina funcione a la perfección.

En esta tercera parte estudiaremos las rutinas correspondientes a la impresión de caracteres o mensajes, (simulación de PRINT, CLS, INK, PAPER, AT...), rutinas de sonido (simulación de BEEP), rutinas de gráficos (simulación de PLOT, DRAW y CIRCLE), y rutinas de cassette (simulación de LOAD y SAVE en un *programa replicante*). A cada una de estas se le dedicará un capítulo, excepto a)a primera, que trataremos a continuación.

RUTINAS DE IMPRESIÓN

Al decir "de impresión" me refiero a la función que ejercen: imprimir caracteres en pantalla, aunque se pueda decir que son bastante impresionantes.

Si no conociéramos ninguna de estas rutinas sería posible afectar directamente a la memoria de pantalla introduciendo ciertos pixels para formar un carácter determinado. Un pixel es cada uno de los puntos que forman un carácter. Este sistema sería bastante largo y difícil de desarrollar.

Pero tenemos la ventaja de conocer algunas rutinas de la ROM, que realizan la impresión de cualquier carácter. Pero antes de entrar en detalles sobre cómo se imprime un carácter determinado, debemos conocer otras cosas.

Un carácter es la representación gráfica de una letra, una cifra u otro símbolo, es la forma en que está escrita. Existen muchos tipos de caracteres. No tenemos más que coger un períodico para comprobarlo.

El único juego de caracteres del que dispone el ZX Spectrum es bastante amplio. Consta de 256 caracteres, numerados de 0 al 255. A cada carácter corresponde un número, denominado código. El juego de caracteres se divide en varias partes:

1. Códigos 0 a 31: caracteres no imprimibles o correspondientes a controles de impresión (INK, PAPER, FLASH...).

2. Códigos 32 a 127: ésta es la parte más importante del juego de caracteres. Corresponde a los caracteres ASCII. Los fabricantes de ordenadores y periféricos se pusieron un día de acuerdo en corresponder unos determinados códigos o números con unos caracteres. Así se creó el American Standard for Information Interchange, usado hoy día universalmente.

El código ASCII 97 corresponde al carácter de "a", el 98 al de "b". El juego de caracteres ASCII del Spectrum difiere sólo en los códigos 96 y 127, que en este ordenador corresponde a los caracteres de Libra Esterlina y el Signo de Copyright.

En el Apéndice A del libro podrás encontrar una tabla con las correspondencias código hex-dec y carácter .

3. Códigos 128 a 143 y 144 a 164: corresponden a los caracteres gráficos. Los primeros son los no definibles. Se pueden encontrar en modo G, pulsando las teclas 1 a 8 con y sin CAPS SHIFT. Los segundos son los caracteres definibles, almacenados a partir de la dirección de memoria definida por la variable del sistema UDG. Se consiguen pulsando las teclas A a U en modo G. Al conectar el aparato, sus caracteres son los de las letras A a U .

4. Códigos 165 a 255: corresponden a los tokens, palabra inglesa con el significado de término simbólico. A ellos pertenecen los gráficos de los comandos y las funciones. Son los que permiten que aparezca la orden completa (por ejemplo, PRINT), al pulsar una sola tecla (por ejemplo la P).

Un carácter ASCII está formado por 8 bytes de información. Cada byte consta de 8 bits, de manera que en total serán 64 bits. Cada uno de estos bits se interpreta como un punto en blanco o en negro, dependiendo de su valor (0 para blanco, 1 para negro). Las diversas combinaciones de puntos blancos y negros formarán todo el juego de caracteres ASCII.

Al imprimir un carácter en pantalla, se modifican 8 direcciones de memoria de la memoria de pantalla, que pasan a contener los valores de cada byte del carácter.

Un detalle muy importante de mencionar es el de la necesidad de tener acceso a la pantalla. Es decir, antes de intentar imprimir un carácter es necesario comprobar si está abierto el canal de la pantalla. Si recuerdas bien, existen 4 canales de información diferente. El número dos es el correspondiente a la pantalla.

Si no le indicamos al microprocesador que queremos que la información salga por el canal 2, existe la posibilidad que él la envíe por el canal 0 y aparezca en la parte inferior del televisor o por el canal 3 y aparezca en la impresora. El canal 3 o puerta hacia la pantalla se "abre" utilizando una subrutina de la memoria ROM llamada "CHAN-OPEN" ("ABRIR CANAL"). Está ubicada en la dirección 1601 h / 5633 d. Se accede con una instrucción CALL. El Acumulador debe contener el canal a abrir, por donde la información va a ser enviada.

Simplemente con estas dos instrucciones se asegura que la información saldrá por pantalla es saber que la subrutina CHAN-OPEN utiliza todos los registros de uso general. De manera que si estos almacenan algún valor que no se quiera perder, sería necesario conservarlo en el Stack con una instrucción PUSH antes de acceder a la subrutina.

Tras este punto podemos pasar a conocer una rutina ROM y utilizarla conjuntamente con la anterior. Esta tiene la función de borrar la pantalla, la misma que el comando CLS. Recibe el nombre de "CLS COMMAND ROUTINE" ("RUTINA DEL COMANDO CLS"). En realidad hace

uso de las rutinas y subrutinas utilizadas por el sistema operativo para ejecutar un CLS. El punto de entrada de la rutina es el 0D6Bh /3435 d. Ningún registro debe contener algún valor especial, como era el caso anterior. Solamente "llamando" a la rutina se ejecutará un CLS. En este caso, como en el anterior, todos los registros de uso general son utilizados. Si se quiere no perder sus contenidos, se deben almacenar, por ejemplo, en la pila de máquina. Esta rutina no necesita tener abierto el canal 2. Esta función ya la realiza la propia rutina.

Esta rutina y la subrutina anterior se suelen usar conjuntamente al principio de numerosos programas en código máquina del ZX Spectrum para tener, desde el principio, acceso a la pantalla y que esté borrada.

Este sería el listado desensamblado:

```
=======================================================================
Especificaciones : "Prog 10.1" para ZX Spectrum 16K/48K.
Descripción General : Se ejecuta un borrado de la pantalla y se
abre el canal 2.
Longitud : 9 bytes.
Entrada : Ningún requerimiento.
Salida : Ningún requerimiento.
Registros Usados : A en programa
                   A,B,C,D,E,H y L en rutinas ROM

7000            00100   ORG 7000H       ;
7000   CD6B0D           00110   CALL CLS        ; BORRA PANTALLA
7002     3E02   00110   LD A,2          ;       ABRE
7005   CD0116           00120   CALL OPEN       ; CANAL NUMERO 2
7008       C9   00140   RET             ;
0000            00150   END             ;

0D6B   CLS
1601   OPEN
=======================================================================
```

Observa que en primer lugar aparece la llamada a la rutina de CLS, seguida la apertura del canal 2. La rutina borra la pantalla en dos tiempos, primero la parte superior de la pantalla (líneas 0 a 21) y después las dos últimas líneas. Para realizar la primera operación, la rutina abre el canal 2, y para realizar la segunda, abre el canal 0, el K. Esto significa que al abandonar la rutina, el canal abierto será el 0, el correspondiente a las dos líneas inferiores, y no a la pantalla. Por esto es necesario colocar la llamada a CHAN-OPEN tras la llamada a CLS. En caso contrario quedaría borrada la pantalla, pero quedaría abierto el canal 0.

La posición donde se imprimirá el carácter será, tras abrir el canal, la correspondiente a la fila 0, columna 0 (0/0), en el extremo superior izquierdo. A continuación aparece el listado BASIC correspondiente al programa 10.1 :

```
10 REM PROG 10.1
20 CLEAR 28650: LET T=0
```

```
 30 FOR F=28672 TO 28680
 40 READ A: POKE F,A
 50 LET T=T+A: NEXT F
 60 IF T<>818 THEN PRINT "ERROR EN DATA": STOP
 70 FOR F=l TO 12: PRINT "-PULSA UNA TECLA PARA PONER EN
    MARCHA EL CODIGO MAQUINA-";: NEXT F
 80 PAUSE 0: RANDOMIZE USR 28672
100 DATA 205 , 107 , 13 , 62 , 2 , 205 , 1 , 22 , 201
```

Pero todavía nos falta la parte más importante. ¿Cómo se imprimirá en el televisor, por ejemplo, un asterisco (*)?

Lo único que hay que saber es la subrutina ROM a la que debemos acceder. Esta se encuentra en la dirección de memoria 15F2h / 5618d. El acumulador contiene el código del carácter a imprimir. El código del carácter "*,, es el 2Ah / 42d. Estos datos son suficiente para poder suplir la orden PRINT en código máquina. El programa assembler sería:

```
=======================================================================
Especificaciones : "Prog 10.2" para ZX Spectrum 16K/48K.
Descripción General : Tras ejecutar un CLS y abrir el canal 2
 , se imprime en la posicion (0/0) un asterisco.
Longitud : 14 bytes.
Entrada : Ningún requerimiento.
Salida : Caracter de "*" en posición (0/0).
Registros Usados : A en este programa
                   A,B,C,D,E,H y L en rutinas ROM

7000            00100   ORG 7000H       ;
7000    CD6B0D          00110   CALL CLS        ; BORRA PANTALLA
7003      3E02  00110   LD A,2          ;       ABRE
7005    CD0116  00120   CALL OPEN       ; CANAL NUMERO 2
7008      3E2A  00140   LD A,42 D       ; CHR$ 42 = "*"
700A    CDF215  00150   CALL PRINT      ; PRINT "*"
700D        C9  00160   RET             ;
0000            00170   END             ;

0D6B CLS
15F2 PRINT
1601 OPEN
=======================================================================
```

La rutina es sencillísima. El acumulador contiene el código del carácter, en este caso, el código 42. La subrutina PRINT se encarga de imprimir el carácter correspondiente. Este es listado BASIC que carga los códigos decimales del programa "Prog 10.2" :

```
 10 REM PROG 10.2
 20 CLEAR 28650: LET T=0
```

```
 30 FOR F=28672 TO 28685
 40 READ A: POKE F,A
 50 LET T=T+A: NEXT F
 60 IF T<>1390 THEN PRINT "ERROR EN DATA": STOP
 70 PRINT " PULSE UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA": PAUSE 0
 80 RANDOMIZE USR 28672
100 DATA 205 , 107 , 13 , 62 , 2 , 205 , 1 , 22
110 DATA 62 , 42 , 205 , 242 , 21 , 201
```

Sin embargo podemos hacerlo todo más fácil. El que creó el ZX Spectrum pensó que esta subrutina se iba a utilizar mucho, por su función y nos facilita su acceso. Recordarás que existían unas instrucciones un tanto especiales, llamadas *RST* (Cap. 7). Hacían uso del direccionamiento por página cero. Estas instrucciones, en sus diferentes formatos, equivalen a unas subrutinas. La gran diferencia y también la gran ventaja es que las instrucciones RST ocupan sólo un byte en memoria. Una de ellas es

RST 10H   Código Objeto   D7

y accede a la dirección de memoria 0010h (16d). En esa dirección de memoria se encuentra la instrucción JP 15F2h (Salta inmediatamente a la dirección de memoria 15F2h).

RST 10H y CALL 15F2H son totalmente equivalentes. Las dos acceden a la rutina de Impresión del contenido del registro A, interpretado como código. Se diferencian en la ocupación de memoria. La primera es más económica, y por lo tanto, mejor.

Modifica tú mismo el programa "Prog 10.2" por otro en el que no aparezca la instrucción CALL 15F2h, si no que quede reemplazada por la instrucción RST 10H.

Probemos ahora a analizar un par de programas más largos donde aparezca la la instrucción CALL 15F2h, si no que quede reemplazada por la instrucción RST 10H.

Probemos ahora a analizar un par de programas más largos donde aparezca la instrucción RST 10H.

Programa "Prog 10.3:

==

Especificaciones : "Prog 10.3" para ZX Spectrum 16K/48K.
Descripción General : Un bucle controla la impresión de todos
los caracteres ASCII de1 Spectrum.
Longitud: 19 bytes.
Entrada : Ningún requerimiento.
Salida: Caracteres ASCII (códigos 32 a 127) en pantalla.
Registros Usados : A en programa
A , D , E , H y L en rutina ROM

```
7000                00100              ORG 7000H        ;
7000      CD6B0D             00110              CALL CLS          ;  BORRA PANTALLA
7003        3E02  00120              LD A,2 D            ;            ABRE
7005      CD0116 00130              CALL OPEN       ; CANAL NUMERO DOS
7008        3E20  00140              LD A,32D          ; PRIMER CODIGO ASCII
700A          F5   00150    SIGUI   PUSF AF             ; ALMACENA CODIGO
700B          D7   00160              RST 10H            ; IMPRIMELO !
700C          F1   00170              POP AF              ; RECUPERA CODIGO
700D        FE80  00180              CP 127D             ; ¿ A = 127 ?
700F          C8   00190              RET Z                ; RETORNO SI A = 127
7010          3C   00200              INC A                ; A=A+1
7011        18F9  00210              JR SIGUI            ; SIGUIENTE CARÁCTER ASCII
0000                00220              END                   ;

700A     SIGUI
================================================================================
```

Este programa tiene la función de imprimir todo el conjunto de caracteres ASCII del Spectrum (códigos 32 a 127) con la instrucción RST 10H.

Tal y como muestra el listado assembler, el programa tiene su comienzo en la dirección de memoria 7000h. En realidad puede ser relocada en cualquier otro lugar, pues utiliza direcciones de memoria relativas (JR SIGUI y no JP SIGUI). Listado BASIC del programa 10.3:

```
 10 REM PROG 10.3
 20 CLEAR 28650: LET T=0
 30 FOR F=28672 TO 28690
 40 READ A: POKE F,A
 50 LET T=T+A: NEXT F
 60 IF T< >2324 THEN PRINT "ERROR EN DATA" : STOP
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL PROGRAMA
    BASIC EQUIVALENTE": PAUSE 0
 80 FOR F=32 TO 127: PRINT CHR$ F;: NEXT F
 90 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA": PAUSE 0
100 RANDOMIZE USR 28672
200 DATA 205 , 107 , 13 , 62 , 2 , 205 , 1 , 22
210 DATA 62 , 32 , 245 , 215 , 241 , 254 , 127 , 200 , 60 , 24 , 247
```

Las tres primeras instrucciones tienen la finalidad de abrir el canal de la pantalla y borrar todo su contenido, como ya hemos visto.

El acumulador se inicial iza con el valor 32d / 20h, que corresponde al primer código del juego de caracteres ASCII del Spectrum. Este registro será utilizado para almacenar el código del carácter que se procese en ese momento. El código 32 corresponde al carácter de espacio.

Es importante saber que la instrucción RST 10H modificará el valor actual de A, cuyo contenido deseamos no perder, pues es el código del carácter a imprimir. Por ello se hace necesario almacenarlo, por ejemplo en la pila de máquina. La instrucción PUSH AF lo guardará allí el tiempo que queramos. También se almacena el registro F, el de los indicadores, pero en realidad eso no tiene ninguna importancia para nosotros.

La instrucción RST puede ejecutarse sin ningún temor. Todo está bajo control. El primer carácter que aparecerá en pantalla será el de espacio, correspondiente al código 32. El segundo será el de exclamación abierta (!), correspondiente al código 33. Observa que éste se imprime junto al anterior, y no bajo él. El resto de los caracteres también serán impresos uno junto al otro. Una vez finalizada esta subrutina, el Acumulador contiene otro valor totalmente diferente al que tenía al empezarla.

Esto no significa ningún problema para nosotros. Sabemos que el valor de A se encuentra "copiado" en el Stack. La instrucción POP AF recupera ese valor, borrando el anterior, que no nos interesaba.

El programa terminará en el momento en que se hayan impreso los 96 caracteres del conjunto ASCII, numerados con los códigos 32 a 127. En ese momento se deberá retornar al BASIC. El Acumulador contiene el código recién procesado. Solamente hace falta comprobar si ha alcanzado el número 127 (último código del conjunto), para saber si hay que retornar. Y esta función es precisamente la que ejerce la instrucción de Comparación. El contenido del registro A, sea cual sea, se compara con la constante 127. En caso que ambos sean iguales, habremos terminado con todos los caracteres ASCII. Si el Acumulador es igual a 127, el indicador de cero tomará el valor 1. En caso contrario, tomará el valor 0. Nótese que CP no modifica en ningún caso el valor del registro A.

Una instrucción de retorno condicional (RET Z) nos devolverá al sistema BASIC si el indicador de cero es igual a 1. Esto último sería la consecuencia que A fuera igual a 127, lo que significaría haber terminado con los caracteres.

En caso diferente, el programa continuará a la línea 00200. El contenido del registro A se incrementa en uno, para que contenga el código del próximo carácter a imprimir. Seguidamente se ejecuta un salto relativo negativo a la parte del programa etiquetada con SIGUI (de Siguiente). El proceso volverá a empezar, pero esta vez con otro código y otro carácter .

El programa imprimirá así los 96 caracteres del conjunto.

El programa BASIC que carga los códigos contiene una rutina escrita en BASIC que realiza la misma función. Comprueba tú mismo la diferencia entre uno y otro.

El segundo programa assembler que será analizado tiene la función de llenar la pantalla con un carácter determinado. Aparte del interés que tiene como programa en código máquina, puede serte útil.

```
=======================================================================
Especificaciones : "Prog 10.4" para ZX Spectrum 16K/48K.
Descripción General : La totalidad de la pantalla se llena con
un solo carácter a alta velocidad.
Longitud : 20 bytes .
Entrada : Ningún requerimiento.
Salida : Pantalla llena con un carácter determinado.
Registros Usados : A , B , C en programa.
                   A , D , E , H y L en rutina ROM.

7000              00100         ORG 7000H       ;
7000    CD6B0D          00110           CALL CLS        ; BORRA LA PANTALLA
7003      3E02    00120         LD A,2D         ;   ABRE
7005    CD0116  00130           CALL OPEN       ; CANAL 2
7008    010003  00140           LD BC,768 D     ; 768=24 FILAS*32 COLUMNAS
700B      3E2A    00150  SIGUI  LD A,42         ; 42 = CODE "*"
700D        D7    00160         RST 10H         ; IMPRIME EL CARACTER "*"
700E        78    00170         LD A,B          ;
700F        B1    00180         OR C            ; B OR C
7010        C8    00190         RET Z           ; SI B = C = 0 ---> RET
7011        0B    00200         DEC BC          ; DECREMENTA CONTADOR
7012      18F7    00210         JR SIGUI        ; SIGUIENTE CARACTER
0000              00220         END             ;

700B    SIGUI
=======================================================================
```

Al utilizar direcciones relativas, y no absolutas, este programa puede ubicarse donde el usuario estime oportuno.

El programa funciona de la siguiente manera: las tres primeras instrucciones son las que cumplen la función de borrar la pantalla y abrir el canal 2, de manera que no haya problemas en la transmisión de los datos.

El registro doble BC contiene el número de veces que el carácter deberá ser impreso. Este registro funcionará como un contador. La pantalla completa (24 líneas) consta de 24 * 32 = 768 posibles caracteres. Son estos los que queremos ocupar.

El código que he escogido es el 42d / 2Ah, correspondiente al carácter de asterisco (*). Si se quiere utilizar otro, no hace falta más que modificar el décimotercer byte del programa. Este es el byte de datos de la instrucción LD A, n, que carga en el acumulador el código del carácter a imprimir. Si en lugar de un asterisco, se desea imprimir el signo de copyright, el treceavo byte será 7Fh / 127d.

La instrucción RST 10H permite imprimir el carácter elegido en la pantalla. Recuerda que la segunda vez se imprimirá junto a éste, de manera que al final se habrán ocupado las 768 posiciones de la pantalla.

Las tres siguientes instrucciones tienen la función de comprobar si el contador ha llegado a su fin. Este método hace uso de la instrucción lógica OR. Esta afecta al indicador de cero, poniéndolo con valor 1, si el resultado de los 2 bytes OReados a sido cero. Esto ocurrirá en el programa cuando los registros B y C sean iguales a cero. Estos se decrementan cada vez que pasan por línea 00200. Un retorno condicional de cero se ejecuta en el momento en que, como consecuencia de la operación OR, el indicador de cero contiene el valor 1.

En caso diferente, como dicho, se decrementará el registro BC. Esta instrucción no afecta a ningún indicador. Tampoco puede producirse un retorno, pues aquella instrucción va antes de la decrementación.

Un salto relativo negativo se ejecuta para enviar el Contador de Programa a la Etiqueta SIGUI (de Siguiente) y continuar con el programa.

Puedes comprobar la diferencia de velocidad en la ejecución comparando esta rutina de código máquina con el programa BASIC que acompaña al cargador, que realiza la misma función.
Este es el listado que corresponde al programa assembler anterior:

```
 10 REM PROG 10.4
 20 CLEAR 28650: LET T=0
 30 FOR F=28672 TO 28691
 40 READ A: POKE F,A
 50 LET T=T+A: NEXT F
 60 IF T<>1909 THEN PRINT "ERROR EN DATA": STOP
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA UN PROGRAMA
    BASIC EQUIVALENTE": PAUSE 0: CLS
 80 FOR F=l TO 704: PRINT "*";: NEXT F: PAUSE 100: CLS
 90 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA": PAUSE 0
100 RANDOMIZE USR 28672
200 DATA 205 , 107 , 13 , 62 , 2 , 205 , 1 , 22
210 DATA 1 , 191 , 2 , 62 , 42 , 215 , 120 , 177 , 200 , 11 , 24 , 247
```

LOS CONTROLES DE IMPRESION. LA RUTINA "PR-STRING"

La primera parte en la división de los códigos en el juego de caracteres del ZX Spectrum pertenece a los códigos 0 a 31.

Algunos de estos 32 códigos tienen una función determinada y pueden ser útiles, y otros no. Los no utilizados por el ordenador, aproximadamente la mitad, son los numerados del 0 al 5, del 24 al 31 y el 15. Estos tienen como carácter una interrogación cerrada.

El carácter correspondiente al resto de los códigos varía. Unos tienen el carácter de interrogación cerrada, otros caracen de un carácter visible y el carácter de otros códigos no puede ser siguiera impreso (Prueba: PRINT CHR$ 16).

Los códigos 16 a 23 se denominan *códigos o controles de impresión*, precisamente por su función, que será estudiada a continuación. Cada uno de ellos puede controlar o afectar al resultado de una impresión, por ejemplo que algo se imprima con un atributo determinado, o en un lugar determinado, o las dos cosas.

Estos códigos pueden ser utilizados en lenguaje BASIC o en código máquina. Conozcámoslos en BASIC:

CODIGO 16 : CONTROL DE TINTA

Este código, seguido de un dato numérico, determina la tinta (INK) con la que se imprimirá uno o varios caracteres, que tienen que venir a continuación en el comando. Es un control de tinta temporal, pues solamente afectará a aquellos caracteres que le sigan en el comando:

```
PRINT CHR$ 16 + CHR$ 3 + "TINTA VIOLETA"
PRINT "TINTA NEGRA"
```

En el ejemplo se ve claramente como el primer mensaje se imprimió en tinta violeta. Esto fue la consecuencia del control de impresión 16. La siguiente orden PRINT vuelve a usar la tinta negra, que supongo era la que tenías en ese momento.

Observa la sintáxis del primer comando. El número de tinta utilizada se determina mediante el carácter correspondiente al mismo número (Tinta 3 = CHR$ 3; Tinta 5 = CHR$ 5 etc.).

Si intentas utilizar una tinta fuera del margen apropiado, el ordenador contestará con el informe "K-Invalid Colour".

CODIGO 17 : CONTROL DE FONDO

Este código, seguido de un dato numérico, determina el fondo (PAPER) que será usado para imprimir una serie de caracteres que deben venir a continuación. Al igual que el anterior, este control es en lenguaje BASIC, temporal:

```
PRINT CHR$ 17 + CHR$ 6 + "FONDO AMARILLO"

PRINT "FONDO BLANCO"
```

La sintaxis es la misma que la usada en el ejemplo anterior. Un número fuera de márgen provoca la aparición del informe "K-Invalid Colour".

CODIGO 18 : CONTROL DE FLASH
CODIGO 19 : CONTROL DE BRIGHT
CODIGO 20: CONTROL DE INVERSE
CODIGO 21 : CONTROL DE OVER

Dada la similitud de estos cuatro códigos de control, éstos pueden ser tratados conjuntamente.

Las funciones de FLASH, BRIGHT, INVERSE y OVER pueden ser accedidas gracias a estos códigos. Tras el código correspondiente a la función que deseemos, debemos introducir el carácter del número que especifique "Apagado" (0), "Encendido" (1), o "Transparente" (8). Esto último sólo puede ser usado con las funciones FLASH y BRIGHT. Los cuatro códigos pueden combinarse entre sí:

```
PRINT CHR$ 18 + CHR$ 1 + CHR$ 19 + CHR$ 1 + CHR$ 20 + CHR$ 1 +
CHR$ 21 + CHR$ 1 + "FLASH 1,BRILLO 1,INVERSE 1,OVER 1"
```

CODIGO 22 : CONTROL DE AT

La función AT es una de las pocas que el ZX Spectrum tiene en exclusiva frente a otros ordenadores domésticos. Permite visualizar caracteres en cualquier parte de la pantalla del televisor, teniendo a disposición 22 líneas y 32 caracteres.

Tras este código deben definirse dos caracteres, correspondientes a dos números, la fila y la columna donde aparecerá el carácter.

```
PRINT CHR$ 22 + CHR$ 11 + CHR$ 16 + "*"
```

Esta sentencia imprime en la fila 11, columna 16, el carácter de asterisco (*). Es por tanto equivalente a PRINT AT 11 ,16; "* "

CODIGO 23 : CONTROL DE TAB

Esta función tiene por objeto determinar la columna a partir de la cual se imprimirá la información. La primera columna donde se puede imprimir algo es la cero, y la última -asómbrate- la 65535. La razón reside en que el ordenador reduce el número de la columna, si esta sobrepasa la número 31 (divide por 32 y se lleva el resto). Por ello TAB 32 es equivalente a TAB 0, TAB 33 a TAB 1 y TAB 65535 a TAB 31 .

Si se utiliza el código 23, la columna se especifica con dos caracteres. Cada uno de ellos debe ser el valor decimal correspondiente a dos cifras hexadecimales de lo que sería la columna expresada en base dieciséis. Imagina que la columna escogida es la 65535. Su correspondiente hex es el FFFFh. Este se puede dividir en dos números decimales, uno conteniendo el byte más significativo y el otro el byte menos significativo. Estos números serían el 255 y el 255 (255 + 256 * 255 = 65535).

La codificación quedaría de la siguiente manera:

```
PRINT CHR$ 23 + CHR$ 255 + CHR$ 255 + "* "
```

El primer carácter corresponde al byte menos significativo, y el segundo al más significativo. De manera que si simplemente deseamos un TAB 10, quedaría así:

PRINT CHR$ 23 + CHR$ 10 + CHR$ 0 + "*"

El segundo carácter corresponde al código 00 que a su vez corresponde al byte más significativo.

Todos estos 8 códigos de control pueden combinarse entre sí. En realidad, la utilización de los caracteres de control en lenguaje BASIC no tiene mucho sentido, pues se dispone de funciones que realizan la misma función.

Pueden desarrollar una gran función en lenguaje ensarnblador. Nos permitiría imprimir cualquier carácter, en cualquier atributo, y en cualquier lugar de la pantalla. Pero ¿cómo introducirlos en programas assembler?

El método más sencillo es tratarlos como simples códigos ASCII. El sistema operativo del ordenador los puede interpretar y ejecutar. La instrucción RST 10H los introduce y procesa.

El siguiente realiza la misma función que el programa BASIC que imprimía en tinta violeta un asterisco, con ayuda de los códigos de control. En código máquina también se hace necesario transmitir al sistema operativo los códigos 16 (control de tinta) y 3 (dato para color violeta).

```
=======================================================================
Especificaciones :. "Progl.0.5" para ZX Spectrum 16K/48K.
Descripción General : Utilización del código de color de tinta.
Longitud : 18 bytes.
Entrada : Ningún requerirniento.
Salida : Impresión de un asterisco en tinta violeta.
Registros Usados: A en Programa
                  A , B , C , D , E , H y L en Rutinas ROM.

7000              00100  ORG 7000H     ;
7000  CD6B0D             00110  CALL CLS      ; BORRA PANTALLA
7003    3E02      00120  LD A,2D       ;       ABRE
7005  CD0116      00130  CALL OPEN     ; CANAL NUMERO 2
7008    3E10      00140  LD A,16D      ; CONTROL DE TINTA ( 16)
700A      D7      00150  RST l0H       ; IMPRIMELO
700B    3E03      00160  LD A,3D       ; DATO:TINTA VIOLETA (3)
700D      D7      00170  RST l0H       ; IMPRIMELO
700E    3E2A      00180  LD A,42D      ; CHR$ 42 ="*"
7010      D7      00190  RST l0H       ; IMPRIMELO
7011      C9      00200  RET           ;
0000              00210  END           ;

0D6B  CLS
1601  OPEN

=======================================================================
```

Las tres primeras instrucciones borran la pantalla y abren el canal 2. El programa empieza realmente a partir de la siguiente instrucción.

La base es hacer que el ordenador imprima con RST 10H un código de control. El o los bytes que le siguen serán interpretados como datos para los controles. Los códigos se cargan en el Acumulador y se procesan con la instrucción RST 10H. El código de tinta (16) necesita de un solo dato: el dato de color. El próximo código que es introducido por RST 10H lleva la significación de Tinta Violeta.

A partir de este código, los siguientes son interpretados por RST 10H como caracteres a imprimir, por supuesto si no son códigos de impresión. El decimoquinto byte del programa contiene el dato para el caracter a imprimir, un asterisco.

El efecto de los códigos de control de color en un programa en código máquina es permanente. En el asterisco ejemplo significaría que los sucesivos RST 10H de códigos ASCII, si los hubiera, imprimirían los caracteres en tinta violeta. El color no cambiaría hasta que se introdujera en el ordenador otro código de control de color, con otro dato. y este es el listado del programa 10.5:

```
 10 REM PROG 10.5
 20 CLEAR 28650: LET T=0
 30 FOR F=28672 TO 28689
 40 READ A: POKE F, A
 30 LET T=T+A: NEXT F
 60 IF T<>1710 THEN PRINT "ERROR EN DATA" : STOP
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
     MAQUINA": PAUSE 0
 80 RANDOMIZE USR 28672
100 DATA 205 , 107 , 13 , 62 , 2 , 205 , 1 , 22
110 DATA 62 , 16 , 215 , 62 , 3 , 215 , 62 , 42 , 215 , 201
```

Como ejercicio, te propongo realizar programas en código máquina que cumplan las mismas funciones que los ejemplos BASIC del principio del apartado.

También deberás escribir un programa en el que aparezcan por lo menos cuatro de los códigos de control y que imprima la palabra "CODIGO MAOUINA" (En BASIC sería PRINT IN K 6; PAPER 1; FLASH 1; BRIGHT 1; AT 11, 8; "CODIGO MAOUINA").

* * *

Supongo que habrás realizado el ejercicio propuesto en las últimas líneas, has comprobado, no la dificultad, sino la longitud, memoria y tiempo que aquel programa propone. Para cada letra se hace uso de 3 bytes, y para cada código de control 2 ó 3, dependiendo de cada uno.

Al que creó la rutina RST 10H le surgió el mismo problema. Si se quiere visualizar mensajes, más o menos largos, y se quiere hacer mediante aquella subrutina de restart, se ocupará bastante memoria, alargando mucho los programas.

Esta es la razón que llevó a crear una segunda subrutina, dedicada especialmente a la impresión de textos, y no de letras o palabras cortas.

Recibe el nombre de PR-STRING (De PRINT STRING, Imprime cadena / texto) y está ubicada en la dirección de memoria 203CH. Es mucho más sencilla de lo que te puedas imaginar (consta sólo de 10 bytes). Con tus conocimientos actuales podías escribirla sin ningún problema.

Para acceder a la rutina se deben cargar los registros con unos valores determinados. El texto debe estar almacenado a partir de una dirección de memoria, pongamos la 7100h, que deben contener los códigos correspondientes a los caracteres que se quieran imprimir, tanto si son ASCII, como de control o gráficos.

El registro doble DE contendrá esta dirección de memoria de comienzo. El registro doble BC contendrá la longitud (en bytes) del texto.

Sólo estos dos datos, comienzo y longitud, serán suficientes para imprimir todo el texto en pantalla y con una sola llamada, un CALL PR-STRING.

El siguiente listado es de un programa ejemplo para PR-STRING. En pantalla se imprime con tinta azul, fondo amarillo, flash y brillo activado, en la posición (11/8) el mensaje "CODIGO MAQUINA":

```
=======================================================================
Especificaciones : "Prog 10.6" para ZX Spectrum 16K/48K.
Descripción General : Impresión de un texto relativamente largo
en pantalIa con la subrutina PR-STRING.
Longitud : 43 bytes .
Entrada : El texto debe estar alrnacenado en forma de códigos a
partir de la dirección 7100h.
Salida : En pantalla el mensaje "CODIGO MAQUINA" , con diversos
códigos de color :y posicionamiento.
Registros Usados : A , B , C , D y E en programa
                   A , B , C , D y E en rutina ROM

7000            00100           ORG 7000H          ;
7000    CD6B0D  00110           CALL CLS           ; BORRA PANTALLA
7003    3E02    00120           LD A,2D            :        ABRE
7005    CD0116  00130           CALL OPEN          ; CANAL NUMERO 2
7008    011900  00140           LD BC,25D          ; LONGITUD DEL TEXTO
700B    110071  00150           LD DE,TEXTO        ; DIRECCION MEMORIA TEXTO
700E    CD3C20  00160           CALL PR-STRING     ; IMPRIME EL TEXTO
7011    C9              00170         RET
7100            00180   TEXTO   ORG 7100H
```

```
7100        1002  00190          DEFB 16,2            ; INK 2
7102        1106  00200          DEFB 17,6            ; PAPER 6
7104        1201  00210          DEFB 18,1            ; FLASH 1
7106        1301  00220          DEFB 19,1            ; BRIGHT 1
7108      160B08  00230          DEFB 22,11,8         ; AT 11,8
710B              00240          DEFM                 ;
                                    "CODIGO MAQUINA"     ;
0000              00250          END                  ;

0D6B    CLS
1601    OPEN
203C    PR-STRING
7100    TEXTO
=======================================================================
```

El listado assembler se analiza de la siguiente manera: las tres primeras líneas tienen la función, como en los últimos programas, de borrar la pantalla y abrir el canal dos.

Seguidamente se introduce en el registro BC la longitud del texto a imprimir (en bytes). El texto consta de una serie de códigos de control de color y posicionamiento de la impresión en pantalla y de las palabras "CODIGO MAQUINA ", Los códigos que forman este texto se encuentran almacenados a partir de la dirección de memoria 7100. El registro doble DE almacena esta dirección. Con estos datos se puede acceder a la rutina PR-STR ING, que los utiliza para imprimir el mensaje.

Una vez terminado esto, se ejecutará la instrucción RET , que devuelve el control al sistema BASIC.

En el listado assembler aparecen dos nuevos términos que aún no he aclarado: DEFB y DEFM. Ambos son dos pseudo-operadores (Pseudo-Opcodes), al igual que ORG y END. Tienen la misión de informar al lector (ya sea un programa ensamblador o una persona). ORG anuncia la dirección de memoria a partir de la cual se encuentran los códigos que vienen a continuación. END indica el fin del programa.

DEFB y DEFM tienen la función de definir los códigos que deben aparecer, si no es por falta de espacio, en la segunda columna. En esto se diferencian de pseudo-operadores como ORG o END, que no se refieren a ningún código objeto. DEFB y DEFM indican al lector que los códigos que aparecen en la segunda columna no se deben interpretar como instrucciones de código rnáquina, sino como datos que serán utilizados por el programa. DEFB especifica que los códigos son bytes (DEFine Byte) de dato, que, como en este caso, tienen una función determinada.

En el listado se definen como bytes los códigos 16 y 2,17 y 6,18 y 1 etc., correspondientes a los controles de color y/o posicionamiento. Si, por ejemplo, el primer byte (16) fuera interpretado como instrucción, se ejecutaría la DJNZ desp, cuando en realidad su función es la de control de tinta. La falta de DEFB en la quinta columna haría confundir la función de los códigos, y seguramente ocasionaría un CRASH.

DEFM indica que el código objeto debe ser interpretado como códigos ASCII, cuyos caracteres serán impresos en un mensaje (DEFine Message). Se relaciona el código objeto con un texto. Normalmente el código objeto no aparece en la segunda columna, por falta de espacio. El mensaje a imprimir aparece en la columna de mnemónicos, entre comillas. A continuación, el listado BASIC correspondiente al programa anterior, el 10.6:

```
 10 REM PROG 10.6
 20 CLEAR 28650: LET T=0
 30 FOR F=28672 TO 28689
 40 READ A: POKE F, A
 50 LET T=T+A: NEXT F
 60 IF T<>1271 THEN PRINT "ERROR EN DATA 200/210" : STOP
 70 LET T=0: FOR F=28928 TO 28952
 80 READ A: POKE F,A
 90 LET T=T+A: NEXT F
100 IF T<>1113 THEN PRINT "ERROR EN DATA 220": STOP
110 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA": PAUSE 0
120 RANDOMIZE USR 28672
200 DATA 205 , 107 , 13 , 62 , 2 , 205 , 1 , 22
210 DATA 1 , 25 , 0 , 17 , 0 , 113 , 205 , 60 , 32 , 201
220 DATA 16 , 6 , 17 , 1 , 18 , 1 , 19 , 1 , 22 , 11 , 8
230 DATA 67 , 79 , 68 , 73 , 71 , 79 , 32 , 77 , 65 , 81 , 85 , 73 , 78 , 65
```

No me voy a ocupar del análisis de la subrutina PR-STRING. Esta materia debe ser realizada por tí mismo. El siguiente programa BASIC "lee" los códigos que la forman y sus direcciones de memoria. Ocúpate tú de encontrar los mnemónicos correspondientes, de la formación del listado assembler y de su estudio.

```
10 FOR F = 8252 TO 8261
20 PRINT F, PEEK F
30 NEXT F
```

11

El Teclado y el Sonido

Este capítulo consta de dos apartados. El primero está dedicado al estudio del teclado. Veremos cómo se puede simular la orden BASIC "IF INKEY$="A" THEN GOTO..." en código máquina. Los dos programas que cierran el libro traen ejemplos prácticos del uso del teclado.

El segundo apartado trata de la generación de sonido en código máquina. Una rutina ROM muy sencilla permite realizar cualquier nota por el altavoz del aparato.

1. EL TECLADO

El teclado es un periférico de entrada de información. Es el canal utilizado más a menudo por el usuario. Otros periféricos de entrada de datos son un cassette, el microdrive o la red de datos del ZX Interface 1.

El teclado consta de 40 teclas, divididas en 4 filas de 10 teclas cada una. En la fila superior se encuentran, entre otras cosas, las diez cifras de nuestro sistema decimal. Las otras 3 filas disponen las teclas igual que una máquina de escribir, incorporando funciones como "ENTER", "CAPS SHIFT", "SYMBOL SHIFT" o "BREAK/SPACE".

La distribución de las teclas es diferente para el ordenador. El divide el teclado en 8 semifilas de 5 teclas cada una. Una semifila es, por ejemplo, la que contiene las teclas 1, 2, 3, 4 y 5. El resto de la fila forma otra semifila, también con 5 teclas.

Cada una de las filas restantes se divide en dos. Si las cuentas no nos engañan, tendremos al final 8 semifilas de 5 teclas, en total 40 teclas.

Existe un mnemónico especial que nos permite acceder al teclado y conocer su estado. Recibe el nombre de IN, y no tiene nada que ver con el comando INPUT del lenguaje BASIC. Las rutinas de ROM del teclado y de LOAD hacen uso de este mnemónico.

Tiene el siguiente formato:

IN A, (N)	Código Objeto	DBXX
IN registro, (C)		

En el primer caso N es una constante numérica y se refiere a un periférico. Cada periférico se accede a través de un "port" (puerta). La constante N define el port a utilizar. Esta instrucción no afecta a ningún indicador. La instrucción IN A, (N) analiza el periférico determinado por la N. El resultado de ese análisis, un byte de datos, es cargado automáticamente en el Acumulador. Dependiendo del periférico, el resultado tendrá una u otra forma.

La instrucción consta de dos bytes. El primero es un código de operación. El segundo es el dato que determina el port a analizar.

En el segundo caso, el número del port está almacenado en el registro simple C. Esta instrucción si que afecta a los indicadores. Consta de dos códigos de operación. Su función es analizar el periférico determinado por el contenido de C. El resultado es cargado en el registro especificado en la instrucción. IN A, (C) cargará el resultado en el Acumulador. El código objeto cambia si se utiliza uno u otro registro.

El número de port utilizado para el teclado es el FEH /254d. La instrucción IN A, (FEh) lee la información del teclado. Pero este es bastante grande, y no puede analizar las 40 teclas a la vez, y volver con un resultado de ellas, almacenado en el acumulador.

Se hace necesario definir la semifila que queremos analizar. Por ello, cada semifila recibe el valor determinado que le hará ser identificada por el ordenador, denominado valor de entrada. Este debe ser introducido en el Acumulador antes de ejecutar la instrucción IN. La semifila que contiene las teclas 1 a 5 recibe el valor de entrada F7H. De esta manera.

LD A,F7H	Código objeto	3EF7
IN A, (FEH)		DBFE

analiza el estado de la semifila 1-5, es decir sí existe alguna tecla pulsada o no. El estado de las teclas se introduce automáticamente en el registro A. El acumulador contiene entonces el byte de dato, o de retorno. Si ninguna tecla de la semifila está pulsada, este byte de retorno, contenido de A, será así:

x x x 1 1 1 1 1

La información sobre las teclas se almacena en los bits. Cada bit almacena el estado de una tecla determinada. Al existir 5 teclas por semifila, son lógicamente 5 los bits que almacenarán la información. Se han escogido los bits 0 a 4. Los de número de orden 5,6 y 7 no guardan ningún valor. En el ejemplo aparecen conteniendo una "x". Con esto se expresa que es indiferente el estado que tengan, 0 o 1.

El bit número 0 (el situado más a la derecha) guarda el estado de la tecla "1 ". El bit 1 guarda el estado de la tecla "2", y así hasta el bit número 4, que guarda el estado de la tecla "5".

Si una tecla no está pulsada, el valor del bit es igual a 1. Por ello, si no hay ninguna tecla pulsada, el valor de retorno es x x x 1 1 1 1 1. En el momento en que alguien ponga un dedo sobre la tecla y la pulse, el valor del bit correspondiente se pondrá a cero.

Si se pulsa la tecla "3", el byte de dato de retorno será igual a x x x 1 1 0 1 1. Si se pulsa la tecla "4", el byte será x x x 1 0 1 1 1. En cada uno de estos dos ejemplos, el bit correspondiente a la tecla contiene el valor 0.

Este valor de retorno nos ofrece la posibilidad de descubrir qué tecla es la que ha sido pulsada. El análisis del estado de los bits nos llevará a esa conclusión.

Antes de realizar un ejemplo práctico, debemos conocer el valor de entrada para cada semifila y el bit que corresponde a cada tecla de cada semifila. La siguiente figura lo muestra:

===

VALOR DE				BITS:		
ENTRADA:		4	3	2	1	0
FEH		V	C	X	Z	Caps Shift
FDH	G	F	D	S	A	
FBH		T	R	E	W	Q
F7H		5	4	3	2	1
EFH		6	7	8	9	0
DFH	Y	U	I	O	P	
BFH		H	J	K	L	ENTER
7FH		B	N	M	Symbol Shift	SPACE/BREAK
		-------	-------	-------	-------	-------
Valor de los ----> bits al pulsar una tecla	11110	11101	11011	10111	01111	

===

La figura nos muestra la posición de cada tecla y el bit que controla su estado. Por ejemplo, la tecla " R ", pertenece a la semifila Q-T , unto a las teclas Q, E, W y T. El valor de entrada de esta semifila es el FBH /251 d. Se encuentra bajo la columna 3, de manera que el bit número 3 del byte de retorno almacena su estado. Si este byte retorna con el valor xxx10111 quiere decir que la

tecla "R" ha sido pulsada durante la ejecución de la instrucción IN, pues el bit n. 3 tiene el valor 0. Como puedes comprobar, cada tecla tiene una semifila, y el estado de la tecla se almacena siempre en un bit. Otra tecla, la de "BREAK/SPACE" tiene como valor de entrada de la semifila el 7FH. Es el bit número 0 el que almacena su estado.

Para saber si ha sido pulsada una tecla, no tenemos más que leer el estado del bit correspondiente. Conocemos dos instrucciones que pueden averiguar el estado de un bit determinado. Estas son la instrucción lógica AND y la instrucción BIT .

El valor de retorno, donde se encuentra toda la información de la fila elegida, es introducido en el Acumu lador tras la ejecución de IN .

Imagina que queremos saber si se ha pulsado la tecla de BREAK/SPACE. En código máquina esta tecla no ejerce su función de para programas. Un programa en código máguina no se puede parar mediante BREAK.

```
LD A, F7H                 Valor de entrada de la semifila
IN A, (FE)                Lee el estado de las 5 teclas
AND 01d                   01d = 00000001 b
        ó
BIT 0,A                   Comprueba el estado del bit cero
JR Z, TECLA PULSADA       Si Indicador de cero = 1 --> Tecla pulsada
```

La instrucción lógica AND realiza una operación AND con el contenido del Acumulador. La constante usada es 16d, cuyo equivalente binario es 00000001b. En caso que la tecla de BREAK no esté pulsada, el resultado de la operación será, con seguridad, diferente de cero.

```
CONTENIDO DEL ACUMULADOR :    x x x 1 1 1 1 1        Tecla no pulsada
AND 01d                  :    0 0 0 0 0 0 0 1
                              ---------------------
Resu1tado diferente de cero   0 0 0 0 0 0 0 1
```

El indicador de cero tomará el valor 0. Pero si la tecla está pulsada, el resultado será también con seguridad igual a cero.

```
CONTENIDO DEL ACUMULADOR :    x x x 1 1 1 1 0        Tecla    pulsada
AND 01d                  :    0 0 0 0 0 0 0 1
                              ---------------------
Resu1tado igual a cero        0 0 0 0 0 0 0 0
```

Y el indicador de cero tomará el valor 1.

Y hacemos uso de la instrucción BIT 0, A el resultado será el mismo. Si el bit a comprobar es igual a 0, el indicador tomará el valor 1. Si el bit a comprobar es igual a 1, el indicador tomará el valor 0.

Gracias al estado del indicador de cero, podemos tomar decisiones. Un salto condicional (por ejemplo, JR Z; salta si el indicador de cero es igual a 1) puede trasladarnos a otra parte del programa, donde se ejecutaría la operación correspondiente a pulsar la tecla.
.

En los siguientes programas utilizaré de la instrucción BIT , cuya función y ocupación de memoria es la misma que AND. Se diferencian en que BIT no destruye el contenido del acumulador. Esto tiene la ventaja que podemos comprobar el estado de una serie de teclas de la misma semifila, sin la necesidad de ejecutar varias veces la instrucción IN.

Podemos utilizar todos estos conocimientos en un pequeño programa, que no devuelva el control al BASIC.hasta que se haya pulsado una tecla determinada, por ejemplo la tecla de BREAK/ SPACE:

Listado assembler "Prog 11.1."

```
=======================================================================
Especificaciones : "Prog 11.1" para ZX Spectrum 16K/48K
Descripción General : Ejernplo del uso del teclado.EI programa no
retorna al BASIC hasta que se pulse la tecla de "BREAK/SPACE".
Longitud : 9 bytes.
Entrada : Ningún requerimiento.
Salida : Ningún requerimiento.
Registros Usados : A

7000       00100          ORG 7000H      ;
7000  3E7F       00110    OTRA  LD A,7FH      ; SEMIFILA B - SPACE
7002  DBFE       00120          IN A, (FE)    ; LEE LA SEMIFILA
7004  CB47       00130          BIT 0,A       ; COMPRUEBA EL BIT CERO
7006  C8         00140          RET Z         ; RETORNO SI BIT = 0 (PULSADA)
7007  18F7       00150          JR OTRA       ; OTRA VEZ,HASTA QUE SEA
                                              ; PULSADA
0000       00160          END            ;

7000       OTRA
=======================================================================
```

El programa utiliza direcciones relativas, de manera que puede ser colocado en cualquier luyar de la memoria.

El acumulador se carga con el valor correspondiente a la semifila escogida (B - SPACE), que es el 7FH. La instrucción IN accede al teclado a la semifila escogida y analiza el estado de las teclas. El resu Itado es introducido automáticamente en el registro A. La instrucción BIT 0, A comprueba el estado der bit número cero, que corresponde a la tecla de BREAK/SPACE. Si la tecla está pulsada, el bit será igual a 0. En este caso, el indicador de cero tomará el valor 1. La instrucción de retorno condicional nos devuelve al BASIC si el indicador es igual a 1. RET se ejecuta en definitiva si se pulsa la tecla de BREAK/SPACE. En caso contrario, el proceso vuelve a

comenzar de nuevo, a partir de la dirección de memoria 7000h. A continuación aparece el listado BASIC correspondiente:

```
 10 REM PROG 11 . 1
 20 CLEAR 28650: LET T= 0
 30 FOR F=28672 TO 28680
 40 READ A: POKE F,A
 50 LET T=T+A: NEXT F
 60 IF T<>1407 THEN PRINT "ERROR EN DATA": STOP
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA " : PAUSE 0
 80 RANDOMIZE USR 28672
100 DATA 62 , 127 , 219 , 254 , 203 , 71 , 200 , 24 , 247
```

Este sistema nos permite realizar en lenguaje máquina la función BASIC de INKEY$, es decir, tomar una decisión determinada de acuerdo a la pulsación de una tecla. El siguiente ejemplo imprime el carácter de asterisco (*) cuando se pulsa la tecla B. El programa vuelve al BASIC al pulsar la tecla N.

```
=========================================================================
Especificaciones : "Prog 11.2"' para ZX Spectrum 16K/48K.
Descripción General : Una asterisco se imprime al pulsar la
tecla B. El programa retorna al BASIC al pulsar la tecla N.
Longitud : 24 bytes .
Entrada : Ningún requerimiento .
Salida : Ningún requerimiento.
Registros Usados : A

7000          00100         ORG 7000H
7000  CD6B0D  00110         CALL CLS      ; BORRA PANTALLA
7003   3E02   00120         LD A,2D       ;        ABRE
7005  CD0116  00130         CALL OPEN     ; CANAL NUMERO 2
7008   3E7F   00140  OTRA   LD A,7FH      ; SEMIFILA B-BREAK/SPACE
700A   DBFE   00150         IN A, (FEH)   ; LEE LA SEMIFILA
700C   CB5F   00160         BIT 3,A       ; COMPRUEBA TECLA "N"
700E    C8           00170         RET Z         ; RETORNA SI PULSADA
700F   CB67   00180         BIT 4,A       ; COMPRUEBA TECLA "B"
7011   20F5   00190         JR NZ,OTRA    ; OTRA VEZ SI NO PULSADA
7013   3E2A   00200         LD A,42D      ; CHR$ 42 = "*"
7015    D7           00210         RST l0H       ; IMPRIMELO
7016   18F0   00220         JR OTRA       ; OTRA VEZ
0000          00230         END

0D6B  CLS
1601  OPEN
7008  OTRA
=========================================================================
```

Este pequeño programa muestra el uso de la instrucción BIT. La lectura de la semifila se realiza solamente en una ocasión. La instrucción BIT no destruye el dato, de manera que puede ser utilizado cuantas veces se quiera.

Se recoge el estado de la semifila B-BREAK/SPACE. En primer lugar se comprueba si el bit número 3, perteneciente a la tecla "N" tiene el valor 0, En ese caso se retornará inmediatamente al BASIC.

En caso contrario y en segundo lugar se realiza la misma operación con el bit número cuatro (tecla B). Si se comprueba que no está pulsado, el programa vuelve otra vez a leer el estado de la semifila. Se retorna a la parte inicial, denominada OTRA.

Si la tecla B está pulsada, se procesa la impresión de un asterisco mediante RST 10H. Tras esta instrucción se vuelve otra vez a leer el teclado. El control del programa pasa a la parte denominada OTRA. Este es el listado BASIC "Prog. 11.2":

```
 10 REM PROG 11.2
 20 CLEAR 28650: LET T= 0
 30 FOR F=28672 TO 28695
 40 READ A: POKE F,A
 50 LET T=T+A: NEXT F
 60 IF T<>2943 THEN PRINT "ERROR EN DATA": STOP
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA": PAUSE 0
 80 RANDOMIZE USR 28672
100 DATA 205 , 107 , 13 , 62 , 2 , 205 , 1 , 22
110 DATA 62 , 127 , 219 , 254 , 203 , 95 , 200
120 DATA 203 , 103 , 32 , 245 , 62 , 42 , 215 , 24 , 240
```

El programa BASIC equivalente sería:

```
10 IF INKEY$="B" THEN PRINT "*" ;
20 IF INKEY$="N" THEN STOP
30 GOTO 10
```

Este apartado se cerrará con un programa que muestra cómo tomar una decisión, no si una tecla está pulsada, sino si dos o más teclas de una semifila se están usando.

Si son dos las teclas de una subrutina las que se pulsan, dos bits tendrán el valor 0. Sólo hace falta modificar la instrucción que nos comprueba el estado de los bits. La instrucción BIT trabaja solamente con un bit. Es en este caso cuando debemos utilizar AND.

La operación AND debe realizarse entre el contenido del acumulador y la suma de los valores que hubieran sido utilizados si se hubieran comprobado los bits separadamente.

El siguiente ejemplo es un programa que, una vez accedido, no vuelve al BASIC, a no ser que se pulsen dos teclas, la P y la Y.

```
==================================================================
Especificaciones : "Progll.3" para ZX Spectrum 16K/48K
Descripción General : El programa no vuelve al BASIC hasta que
se hayan pulsado dos teclas.
Longitud : 9 bytes.
Entrada : Ningún requerimiento.
Salida : Ningún requerimiento.
Registros Usados : A

7000      00100          ORG 7000H      ;
7000  3EDF     00110    OTRA   LD A,DFH       ; SEMIFILA P-Y
7002  DBFE     00120           IN A, (FEH)    ; LEE LA SEMIFILA
7004   E611    00130           AND 17D        ; COMPRUEBA BITS 0 Y 4
                                              ; 17D = 00010001 B
7006    C8     00140           RET Z          ; VUELVE SI ESTAN PULSADAS P E Y
7007  18F7     00150           JR OTRA        ; OTRA VEZ,HASTA QUE ESTEN
0000      00160          END            ; PULSADAS

7000 OTRA
==================================================================
```

Si hubiéramos querido comprobar el estado de la tecla Y, hubiera sido necesaria la instrucción AND 16d. Si hubiéramos querido comprobar el estado de la tecla P, hubiera sido necesaria la instrucción AND 1 d. Pero al querer comprobar el estado de ambas teclas, debe aparecer la instrucción AND 17d.

El número decimal 17 tiene la forma binaria 00010001. La operación AND dará solamente el resultado cero que afectará al indicador de cero, si el byte de retorno tiene el bit número 0 y 4 con valor 0. Esto último significaría que las teclas Y y P han sido pulsadas. Un retorno condicional de cero devuelve el control al BASIC si se pulsan estas teclas.

El programa permanece en un bucle cerrado, hasta que se cumpla la condición de la instrucción AND. y este es el listado BASIC que carga los códigos del programa anterior:

```
 10 REM PROG 11.3
 20 CLEAR 28650: LET T=0
 30 FOR F=28672 TO 28680
 40 READ A: POKE F,A
 50 LET T=T+A: NEXT F
 60 IF T<>1476 THEN PRINT "ERROR EN DATA": STOP
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA": PAUSE 0
 80 RANDOMIZE USR 28672
100 DATA 62, 223 , 219 , 254 , 230 , 17 , 200 , 24 , 247
```

2. EL SONIDO

Una de las mayores peculiaridades del ZX Spectrum sobre su antecesor, el ZX-81 es la incorporación del sonido. El Spectrum posee un solo canal de sonido. Por un pequeño altavoz, situado en la parte inferior de la carcasa del ordenador, se envían sonidos.

En lenguaje BASIC, los sonidos se crean mediante el comando BEEP. BEEP necesita de dos parámetros: duración de la nota, y tono a ejecutar. Ambos se representan mediante dos argumentos numéricos. Para cada nota existe un número determinado. El 0 para el DO CENTRAL, el 1 para el DO SOSTENIDO, el 2 para el RE. Números negativos también pueden ser tratados. Estos representan entonces escalas inferiores. El -1 es el tono SI (una escala inferior), el -2 es el tono LA SOSTENIDO.

El altavoz es uno de los periféricos del ordenador. La instrucción que conocimos en el apartado anterior, IN, recoge información de un periférico. Otra instrucción OUT realiza la función inversa. OUT envía información hacia un periférico, como pueda ser el altavoz.

Esta instrucción es precisamente la que utiliza la rutina de generación de sonido de la memoria ROM. Recibe el nombre de BEEPER y puede ser accedida por el usuario. Utilizando la rutina BEEPER podríamos realizar cualquier tipo de sonido que se pudiera hacer con el comando BEEP.

La rutina BEEPER tiene su comienzo fijado en la dirección de memoria 03B5H /949 d. Esta rutina es realmente fácil de dominar. Sólo es necesario saber qué registros deben mantener los valores necesarios para la realización del sonido. Propongámonos escribir un programa en código mágquina que ejecute el DO central durante 2 segundos. Antes de acceder a BEEPER, los siguientes registros deben contener:

Registro DE	=	frecuencia * duración , en segundos
		= frecuencia * 2
Registro HL	=	(437.500 / frecuencia) - 30,125

Esto se debe a que la rutina BEEPER hace sus propios cálculos para averiguar duración y tono del sonido. Es algo que no podemos modificar. Si queremos utilizar la rutina BEEPER, tenemos que hacer nuestros cálculos previos.

La única variable que hay que despejar es la frecuencia perteneciente al tono DO. Esta es 261,6 Hz. Los registros sólo almacenan valores enteros, de manera que estos serán:

Registro DE	=	261,6 * 2 = 523,2 = <u>523 = 020BH</u>
		= (437.500 /261,6) - 30,125
Registro HL	=	1642,275 = <u>1642 = 066A</u>

El programa assembler sería el siguiente:

```
===============================================================================
Especificaciones : "Prog 11.4" para ZX Spectrum 16K/48K
Descripción General : El tono DO central se realiza durante 2
segundos .
Longitud: 10 bytes.
Entrada : Ningún requerirniento
Salida : Ningún requerimiento
Registros Usados : D,E,Hy L en programa
                   A , B , C , D , E , H , L e IX en rutina

7000            00100   ORG 7000H       ;
7000   216A06   00110   LD HL,1642 D    ; ( 437.50-0 / 261,6 ) - 30,125
7003   110B02   00120   LD DE,523 D     ;  261,6 * 2
7006   CDB503   00130   CALL BEEPER     ; REALIZA EL SONIDO
7009     C9     00140   RET             ;
0000            00150                   ;

03B5    BEEPER
===============================================================================
```

La rutina BEEPER modifica el valor de todos los registros. Si incluyes esta rutina en algún programa tuyo, recuerda que los contenidos importantes de los registros se perderán al acceder a ella. Deberán ser almacenados en alguna parte antes de la instrucción CALL BEEPER y recuperados tras ella (por ejemplo, en el Stack, con las instrucciones PUSH y POP). Este es el listado BASIC correspondiente al programa "Prog 11.4 ":

```
 10 REM PROG 11.4
 20 CLEAR 28650: LET T= 0
 30 FOR F=28672 TO 28681
 40 READ A: POKE F, A
 50 LET T=T+A: NEXT F
 60 IF T< >765 THEN PRINT "ERROR EN DATA" : STOP
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA" : PAUSE 0
 80 RANDOMIZE USR 28672
100 DATA 33 , 106 , 6 , 17 , 11 , 2 , 205 , 181 , 3 , 201
```

Para realizar otro sonido, sería necesario modificar ambos registros dobles. Ahora se presenta la dificultad de averiguar las frecuencias pertenecientes a cada tono. He realizado una tabla que contiene las frecuencias de los tonos que pertenecen a los argumentos numéricos BASIC -12 a +40, poco más de 5 escales. Estos son los sonidos con los que se puede construir alguna que otra melodía. Los valores superiores son muy agudos y los inferiores suenan como chasquidos.

Bajo la columna "VALOR PARA DE" se encuentra el valor que tendría el registro doble DE con la frecuencia correspondiente al tono de la segunda columna y la duración de 1 segundo.

Los valores se dan en sistema hexadecimal y decimal. Este último no es el valor entero, sino redondeado a una décima, por si quieres calcular el valor de DE en duraciones superiores a un segundo.

Bajo la columna "VALOR PARA HL" se encuentra, ya calculado, el valor que se debe introducir en el registro doble HL. Este valor no depende de la duración del tono. Al contrario del valor contenido en DE, el de HL es siempre el mismo para el mismo tono.

===

TABLA DE FRECUENCIAS

NUMERO BASIC	TONO	VALOR PARA DE		VALOR PARA HL	
		HEX	DEC	HEX	DEC
-12	DO	0082	130,6	0CF8	3320
-11	D0#-REb	008A	138,4	0C3A	3130
-10	RE	0092	146,8	0B86	2950
-9	RE#-MIb	009B	155,7	0ADC	2780
-8	MI	00A5	165,1	0A3C	2620
-7	FA	00AE	174,6	09AB	2475
-6	FA#-SOLb	00BA	186,2	0910	2320
-5	SOL	00C4	196,2	0898	2200
-4	SOL#-LAb	00CF	207,8	081B	2075
-3	LA	00DC	220,9	079E	1950
-2	LA#-SIb	00E8	232,7	073A	1850
-1	SI	00F7	247,2	06CC	1740
0	DO	0105	262,6	066A	1642
1	DO#-REb	0115	277,2	060C	1548
2	RE	0125	293,7	05B3	1459
3	RE#-MIb	0137	311,1	0560	1376
4	MI	0149	329,6	0511	1297
5	FA	015D	349,2	04C6	1222
6	FA#-SOLb	0171	367,0	0480	1152
7	SOL	0188	392,0	043D	1085
8	SOL#-LAb	019F	415,3	03FF	1023
9	LA	01B8	440,0	03C4	964
10	LA#-SIb	01D2	466,2	038C	908
11	SI	01ED	493,9	0357	855
12	DO	020B	523,9	0325	805
13	DO#-REb	0229	553,7	02F8	760
14	RE	024B	587,1	02CB	715
15	RE#-MIb	0270	624,9	029E	670
16	MI	0294	660,8	0278	632
17	FA	02BA	698,7	0254	596
18	FA#-SOLb	02DF	735,1	0235	565
19	SOL	030D	781,0	0212	530
20	SOL#-LAb	0341	833,1	01EF	495

21	LA	0371	881,9	0lD2	466
22	LA#-SIb	03A6	934,6	01B6	438
23	SI	03DD	989,5	019C	412
24	DO	0416	1046,3	0184	388
25	DO#-REb	0453	1107,2	016D	365
26	RE	0497	1175,7	0156	342
27	RE#-MIb	04DA	1242,5	0142	322
28	MI	0525	1317,3	012E	302
29	FA	056C	1388,3	011D	285
30	FA#-SOLb	05CA	1482,4	0109	265
31	SOL	0619	1561,8	00FA	250
32	SOL#-Lab	067E	1662,7	00E9	233
33	LA	06E3	1763,2	00DA	218
34	LA#-SIb	073C	1852,8	00CE	206
35	SI	07B1	1969,6	00C0	192
36	DO	0822	2082,1	00B4	180
37	DO#-REb	08AB	2219,4	00A7	167
38	RE	092E	2350,6	009C	156
39	RE# -MIb	09B4	2484,0	0092	146
40	MI	0A49	2633,6	0088	136

===

Los tonos graves, los primeros de la tabla, tienen un valor pequeño en el registro DE y alto en el HL. A medida que se escogen tonos más agudos, el valor de DE aumenta y el de HL disminuye.

Si queremos ejecutar más de una melodía, debemos escribir tantas ordenes LD DE, valor, LD HL, valor y CALL BEEPER como número de tonos tenga. Esto nos llevaría a una cantidad inmensa de instrucciones, que ocuparía mucho espacio en memoria (9 bytes por tono). Analiza la rutina PR-STRING mencionada en el capítulo anterior y, basándote en ella, crea una rutina que vaya leyendo los valores para cada nota y acceda a BEEPER para ejecutarla. Estos valores deberán estar almacenados a partir de una dirección de memoria determinada. Recuerda que la rutina BEEPER modifica el contenido de todos los registros, de manera que si alguno almacena algún valor importante, debe ser guardado.

Utiliza la misma melodía que aparecen en el manual de instrucciones que acompaña a cada ZX Spectrum, en el capítulo 19, pág. 135, llamada "Frere Jacques"

Esta será seguramente tu oportunidad para demostrar tus conocimientos en código máquina y tu capacidad de análisis y programación. No es recomendable continuar con los siguientes capítulos si no se ha pasado esta pequeña "prueba"'

No sólo se pueden conseguir tonos musicales con la rutina BEEPER, sino también efectos sonoros. El siguiente programa es un ejemplo de un "disparo de láser":

===

Especificaciones : "Prog 11.5" para ZX Spectrum 16K/48K.
Descripción General : La rutina BEEPER realiza un sonido

comparab1e a1 disparo de un arma de láser.
Longitud : 21 bytes.
Entrada : Ningún requerimiento.
Salida : Ningún requerimierlto.
Registros Usados : B , C , D , E , H y L en programa
A , B , C , D , E , H , L e IX en rutina ROM

```
7000              00100          ORG 7000H      ;
7000    210500    00110          LD HL,5 D      ; TONO MUY AGUDO
7003    110500    00120          LD DE,5 D      ; DURACION Y FRECUENCIA BAJAS
7006    06A0      00130          LD B,160D      ; CONTADOR
7008    E5        00140   OTRA   PUSH HL        ; GUARDA HL EN STACK
7009    D5        00150          PUSH DE        ; GUARDA DE EN STACK
700A    C5        00160          PUSH BC        ; GUARDA BC EN STACK
700B    CDB503    00170          CALL BEEPER    ; REALIZA SONIDO
700E    Cl        00180          POP BC         ; RECUPERA BC
700F    D1        00190          POP DE         ; RECUPERA DE
7010    El        00200          POP HL         ; RECUPERA HL
7011    23        00210          INC HL         ; INCREMENTA REGISTRO HL
7012    10F4      00220          DJNZ OTRA      ; DECREMENTA REGISTRO Y
                                                       ; REALIZA LA OPERACION DE
                                                       ; NUEVO SI B < > 0
7014    C9        00230          RET            ;
0000              00240          END            ;

03B5    BEEPER
7008    OTRA
=======================================================================
```

En los registros DE y HL se introducen valores muy bajos. La duración del sonido será por tanto corta y su tono, agudo.

El registro B es usado como contador de la operación, que ha de realizarse 160 veces. Siempre que se realiza, se incrementa el valor de HL, de manera que el tono del sonido sea ascendente.

Antes y después del acceso a la rutina BEEPER se almacenan los contenidos de HL, DE y B en el Stack. Estos no se deben perder, pues cada uno de ellos guarda un importante valor.

El contador es controlado por la instrucción DJNZ, que decrementa el contenido del registro B, y si es diferente de cero, repite de nuevo la operación, enviando el programa a la parte etiquetada con "OTRA".

Otros efectos sonoros pueden ser conseguidos modificando tanto los contenidos iniciales de HL, como de DE o B.

Este es el listado BASIC que corresponde al programa anterior:

```
 10 REM PROG 11.5
 20 CLEAR 28650: LET T=0
 30 FOR F=28672 TO 28692
 40 READ A: POKE F,A
 50 LET T=T+A: NEXT F
 60 IF T< >2367 THEN PRINT "ERROR EN DATA" : STOP
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA": PAUSE 0
 80 RANDOMIZE USR 28672
100 DATA 33 , 5 , 0 , 17 , 5 , 0 , 6 , 150
110 DATA 229 , 213 , 197 , 205 , 181 , 3 , 193 , 209 , 225
120 DATA 35 , 16 , 244 , 201
```

12

Comandos Gráficos

En este capítulo nos introduciremos en cómo usar los comandos gráficos PLOT, DRAW y CIRCLE en lenguaje ensamblador .

Estas instrucciones afectan directamente a la memoria de pantalla. Es indispensable conocer a fondo esta zona de la memoria RAM, por eso te recomiendo repasar, en el capítulo dedicado a la memoria RAM, el apartado que recoge este tema.

1. PLOT

La orden BASIC PLOT afecta directamente el archivo de pantalla. PLOT x, y imprime un pixel en las coordenadas x, y. Tras ejecutar una orden PLOT, un bit de un byte, determinado por las coordenadas x e y, tomará el valor 1. PLOT hace uso de la alta resolución del aparato.

Las coordenadas deben permanecer dentro de unos parámetros determinados. La primera coordenada, x, debe estar incluida en el rango 0-175. Esta coordenada determina la fila del pixel. La segunda determina la columna que ocupará el pixel. Esta coordenada y se moverá entre 0 y 255.

En total disponemos de 176 puntos verticales y 256 horizontales. Mediante PLOT tendremos acceso a 176 * 256 = 45056 pixels.

A diferencia de la memoria de pantalla, cuyo primer elemento corresponde al byte del carácter superior izquierdo, y de la baja resolución, cuya posición para el carácter de fila 0, columna 0 es también el extremo superior izquierdo, a la hora de trabajar con comandos gráficos (PLOT , DRAW, CIRCLE) y alta resolución, el primer elemento (fila 0, columna 0) corresponde a la posición extrema inferior izquierda.

Hay que resaltar que la orden PLOT no puede trabajar con las dos líneas inferiores (22 y 23) reservadas para los informes e INPUTs.

Sería posible elaborar un programa ensamblador que, de acuerdo a dos coordenadas x e y, calculase la dirección de memoria correspondiente en el archivo de pantalla e introdujese en la misma un pixel con valor 1.

El razonamiento es relativamente sencillo, pero el algoritmo para pasar a la coordenada alta a coordenada baja es más complicado. Tenemos una solución mucho más sencilla. Siempre es más fácil hacer uso de lo que existe, antes de hacerlo nosotros mismos.

En la memoria de sólo lectura se encuentran todas las rutinas que ejecutan las órdenes BASIC. y también se encuentra la rutina que ejecuta el comando PLOT .

El listado assembler siguiente es precisamente un ejemplo del uso de la rutina PLOT de la ROM:

Listado Assembler "Prog 12.1 "

```
==========================================================================
Especificaciones: "Prog 12.1" para ZX SPECTRUM 16K/48K.
Descripción General : Impresión de un pixel en las coordenadas
 128,88.
Longitud : 8 bytes .
Entrada : Ningún requerimiento.
Salida : Impresión de un pixel en coordenadas B,C
Registros Usados: B, C en prograrna
                  A , B , C , D , E , H , L en rutina ROM

7000            00100   ORG 7000H       ;
7000      0658  00110   LD B,88D        ; COORDENADA VERTICAL
7002      0E80  00120   LD C,128D       ; COORDENADA HORIZONTAL
7004    CDE522          00130   CALL PLOT       ; PLOT 128,88
7007        C9  00140   RET             ;
0000            00150   END             ;

22E5H   PLOT
==========================================================================
```

Las coordenadas correspondientes al pixel deben ser introducidas en los registros B y C. El primero debe contener la coordenada vertical (rango 0 a 175). El segundo la coordenada horizontal (rango 0 a 255).

Una vez contengan ambos registros los valores apropiados, se "llama" directamente a la rutina PLOT, cuyo punto de entrada es la dirección de memoria 22E5H (8933 d).

En la propia rutina PLOT se hace uso de los registros A, D, E, H y L. Si quisieras introducir esta rutina en alguno de tus programas y utilizarás alguno de ellos, deberías almacenarlo, bien en el STACK (PUSH), bien en una dirección de memoria (mediante direccionamiento extendido: LD (DIRECCION), A u otro registro).

En la rutina ROM se encuentra el algoritmo para trasladar las coordenadas de alta resolución a otras de baja resolución.

Precisamente para ello se hace uso del resto de los registros. La rutina se encarga de imprimir el pixel en el lugar deseado, con los atributos definidos en ese momento.

En caso de introducir en los registros B o C un valor fuera del rango, el sistema no se destruye, sino que, como en lenguaje BASIC, aparecería el informe "B-Integer out of range". Este es el listado BASIC que carga los códigos decimales del programa "Prog 12.1":

```
10 REM PROG 12.1
20 CLEAR 28650: LET T=0
30 FOR F=28672 TO 28679
40 READ A: POKE F,A
50 LET T=T+A: NEXT F
60 IF T< >905 THEN PRINT "ERROR EN DATAS" : STOP
70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
   MAQUINA"
80 PAUSE 0: RANDOMIZE USR 28672
90 DATA 6 , 88 , 14 , 128 , 205 , 229 , 34 , 201
```

Existe un sistema para ahorrar memoria, que en ciertas ocasiones puede representar el final feliz de un programa.

Si introducirmos valores a dos registros, que podrían estar emparejados, en este caso los registros B y C, no hay por qué hacerlo separadamente. Existe un mnemónico para dar a un registro doble un valor, en el rango 0-65535. Esto equivale a dar al registro que almacena el byte "bajo" un valor y al que almacena el byte "alto" otro valor. Los registros B y C se emparejan en el registro doble BC. En el caso de "Prog 12.1" habría que usar el mnemónico LD BC, NNNN (valor entre 0 y 65535) .

El listado assembler "Prog 12.2" muestra este cambio, junto con otra modificación:

===
Especificaciones : "Prog 12.2" para ZX Spectrum 16K / 48K
Descripción General : Irnpresión de un pixe1 en 1as coordenadas
128,88.
Longitud : 6 bytes .
Entrada : Ningún requerimiento.
Salida : Irnpresión de un pixel en coordenadas B , C
Registros Usados : B , C en prograrna
A , B , C , D , E , H , L en rutina ROM

```
7000                00100   ORG 7000H
7000        018058 00110    LD BC,22656 d    ; B=58H=88D      C=80H=128D
7003       C3E522           00120    JUMP PLOT        ; PLOT 128,88
0000                00130   END

22E5H    PLOT
======================================================================
```

La rutina da al registro BC el valor 22656 d (5880H), que equivale plenamente a introducir en el registro B el valor 88 d (58H), y en el registro C el valor 128 d (80H).

El ahorro de memoria en este cambio es de un byte. A continuación, el listado BASIC del programa anterior:

```
10 REM PROG 12.2
20 CLEAR 28650: LET T=0
30 FOR F=28672 TO 28677
40 READ A: POKE F, A
50 LET T=T+A: NEXT F
60 IF T< >675 THEN PRINT "ERROR EN DATAS": STOP
70 PRINT " PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
   MAQUINA"
80 PAUSE 0: RANDOMIZE USR 28672
90 DATA 1 , 128 , 88 , 195 , 229 , 34
```

La otra diferencia estriba en realizar un JUMP en lugar de un CALL, al tener acceso a la rutina ROM PLOT.

Si optamos por un CALL, deberá existir una instrucción tras ella, pues al término de la rutina, se RETornará a la instrucción tras la que fue llamada. Si tras el mnemónico CALL existe una instrucción de RETorno incondicional (RET), podemos usar una orden JUMP.

Al término de la rutina que llamemos, tiene que existir una instrucción de RETorno. Si accedemos a ella por medio de un salto (JUMP) y no de una llamada (CALL), nos evitaremos nuestra instrucción RET, pues la misma que existe en la rutina, servirá para RETornar al BASIC.

Una de las facultades de la orden BASIC PLOT es poder hacer uso de funciones tales como OVER o INVERSE. La más interesante de estas dos es OVER 1. PLOT OVER 1; x, y imprimirá un pixel en las coordenadas x, y si anteriormente no existía ninguno. Si esas coordenadas estaban ocupadas por un pixel, éste será borrado.

La realización de esta función en código máquina es relativamente sencilla. Son diferentes variables del sistema las que controlan los atributos permanentes o temporales. Por atributos permanentes entendemos los colores en curso. Nada más conectar el ordenador, éstos serán:

FLASH 0 , BRIGHT 0 , PAPER 7 , INK 0

Los atributos temporales son los que se utilizan únicamente para una sentencia, por ejemplo PRINT INK 4; "ABC".

Cada bit o grupo de bits, según la variable y la función controlan una función determinada.

La variable del sistema P-FLAG (23697 D / 5C91 H) contiene los indicadores OVER e INVERSE. Son los dos primeros bits de esta variable (bits 0 y 1) los que almacenan la información para la función OVER. No tenemos más que modificar el valor de estos bits para acceder a OVER 1 u OVER 0.

Si el indicador OVER 0 está activado, los bits 0 y 1 de la variable del sistema P-FLAG tendrán el valor 0.

Si el indicador OVER 1 está activado, los bits 0 y 1 de la variable del sistema P-FLAG tendrán el valor 1.

Una solución muy sencilla sería introducir mediante direccionamiento extendido el valor en esta variable. Este listado assembler lo muestra:

```
=======================================================================
Especificaciones : "Prog 12.3" para ZX SPECTRUM 16K/48K.
Descripción General : Impresión de un pixel en las coordenadas
 128,88 con la función OVER 1 (Direccionamiento extendido)
Longitud : 17 bytes.
Entrada : Ningún requerimiento.
Sa1ida :   Impresión OVER 1 de un pixe1 en coordenadas B, C
           Variable P-FLAG con valor 0
Registros Usados : A, B, C en programa
                   A , B , C , D , E , H , L en rutina ROM

7000             00100   ORG 7000H        ;
7000      3E03   00110   LD A, 00000011B ; 00000011B = 3 D
7002    32915C   00120   LD (P-FLAG),A    ; ACCESO A OVER 1
7005    018058   00130   LD BC,22656 D    ; B=58H=88D      C=80H=128D
7008    CDE522   00140   CALL PLOT        ; PLOT OVER 1;128,88
700B      3E00   00150   LD A,0           ;
700D    32915C   00160   LD (P-FLAG) ,A   ; ACCESO A OVER 0
7010        C9   00170   RET              ;
0000             00180   END      ;

22E5H   PLOT
5C91H   P-FLAG
=======================================================================
```

El direccionamiento extendido introduce en la variable P-FLAG el valor del registro A, que se inicializa con 3d. El equivalente binario de 3d es 00000011 B. Al introducir esto en la variable, estamos dando valor 1 a los bits 0 y 1. De esta manera, accedemos a la función OVER 1.

Una vez introducidas las coordenadas en B y C, la variable P-FLAG se restaura a 0, tal y como era al principio. Para ello se usa el mismo método, direccionamiento extendido. Este es el listado BASIC del programa 12.3:

```
 10 REM PROG 12.3
 20 CLEAR 28650: LET T=0
 30 FOR F=28672 TO 28688
 40 READ A: POKE F,A
 50 LET T=T+A: NEXT F
 60 IF T< >1587 THEN PRINT "ERROR EN DATAS" : STOP
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA"
 80 PAUSE 0: RANDOMIZE USR 28672
100 DATA 62 , 3 , 50 , 145 , 92
110 DATA 1 , 128 , 88 , 205 , 229 , 34
120 DATA 62 , 0 , 50 , 145 , 92 , 201
```

Sin embargo, este programa puede evolucionar aún más. Cuando se trataron los registros del microprocesador, mencioné uno que recibía el nombre de registro IY. Este, al igual que el IX sólo puede ser tratado como registro doble. El usuario tiene cierta libertad a la hora de usar el registro IX, pero no debe modificar el valor de IY.

Este último mantiene siempre el valor 23610 d / 5C3A H, que es la dirección correspondiente a la variable del sistema ERR NR. Estos registros suelen ser usados como punteros. Pueden estar acompañados de un desplazamiento (-128, + 127) , para acceder directamente a un elemento determinado.

Mediante el direccionamiento extendido y el registro IY, podemos acceder a la variable P-FLAG. Esta última ocupa la dirección 23697 d/5C91 H y el registro IY mantiene su valor 23610 d/5C3A H. El desplazamiento debe ser +87d / +57H. El listado assembler siguiente utiliza este sistema:

```
======================================================================
Especificaciones : "Prog 12.4" para ZX SPECTRUM 16K / 48K
Descripción Genera1 : Impresión de un pixe1 en 1as coordenadas
 128,88 cor, la función OVER 1 (Direccioramiento indexado) .
Longitud : 15 bytes.
Entrada : Ningún requerimiento.
Salida : Irnpresión OVER 1 de un pixel en coordenadas B , C.
Variable P-FLAG con valor 0.
Registros Usados : B , C, IY en programa.
                   A , B , C , D , E , H , L en rutina ROM

7000            00100   ORG 7000H         ;
7000   FD365703         00110   LD (IY+87),11B   ; ACCESO A OVER 1
```

```
7004      018058  00130   LD BC,22656 D   ; B=58H=88D    C=80H=128D
7007      CDE522  00140   CALL PLOT       ; PLOT OVER 1;128,88
700A    FD365700          00160   LD (1Y+87) ,0    ; ACCESO A OVER 0
7010        C9    00170   RET             ;
0000              00180   END             ;

22E5H   PLOT
=================================================================
```

La dirección de memoria resultante de sumar el registro IY y el desplazamiento +87d es precisamente la variable del sistema P-FLAG. Nótese que I Y no es modificado.

Este registro es normalmente usado si quiere modificar o acceder a una variable del sistema determinada. Y para introducir los códigos decimales del programa anterior, teclea el siguiente listado:

```
 10 REM PROG 12.4
 20 CLEAR 28650: LET T= 0
 30 FOR F=28672 TO 28686
 40 READ A: POKE F,A
 50 LET T=T+A: NEXT F
 60 IF T<>1677 THEN PRINT "ERROR EN DATAS" : STOP
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA"
 80 PAUSE 0: RANDOMIZE USR 28672
100 DATA 253 , 54 , 87 , 3
110 DATA 1 , 128 , 88 , 205 , 229 , 34
120 DATA 253 , 54 , 87 , 0 , 201
```

Tras este recorrido por los métodos más elementales de ejecutar la orden PLOT en código máquina, te propongo un ejercicio:

PLOTear un solo punto en lenguaje assembler es relativamente sencillo, como ya hemos visto. Ahora debes PLOTear una serie de puntos. Piensa que si introduces cada vez los valores en B y C, y llamas cada vez a la rutina PLOT, va a ser muy largo y aburrido. Existen otros métodos (Pista: Básate en la rutina PR-STRING).

2. DRAW

La orden BASIC DRAW tiene el formato DRAW x, y. Este comando traza una línea recta entre el último punto PLOTeado y otro, distanciado x pixels horizontalmente e y pixels verticalmente. La sentencia DRAW determina la longitud y dirección de la recta, pero no su comienzo.

En caso de no hacer ejecutado una orden PLOT, DRAW, CIRCLE o también tras un CLS,

RUN, CLEAR o NEW, el Spectrum define como pixel de partida el de coordenadas (0/0), en el extremo inferior derecho.

Los parámetros que pueden adoptar x e y varían desde -255 a +255 para el x, y desde -175 a + 175 para y. Dependiendo de la posición del último punto que fue PLOTeado, la sentencia DRAW será aceptada y ejecutada, o invalidada con Error "B-lnteger out range".

Tenemos dos posibilidades para ejecutar la orden DRAW en código máquina:

(1) Elaborar un algoritmo que simule en lenguaje ensamblador lo que se realiza en BASIC. Este algoritmo debe calcular las distancias absolutas entre pixel y pixel, y la inclinación de la recta con respecto al eje x. De acuerdo a estos datos se PLOTearían los pixels correspondientes.

(2) Utilizar las rutinas de la memoria ROM, que realizan la orden DRAW. Sin duda es esta segunda opción la más sencilla para todos. Precisamente el algoritmo que tendríamos que elaborar, si eligieramos la primera opción, ya está realizado en las rutinas DRAW, y es mucho mejor usar lo que ya disponemos, antes de hacerlo nosotros mismos.

Para poder utilizar esta rutina, que ejecutaría lo mismo que el comando BASIC DRAW, es necesario conocer qué registros tienen que almacenar los valores oportunos para la realización de la recta.

El registro B tiene que almacenar el *valor absoluto* de la coordenada y. El registro C debe almacenar el *valor absoluto* de la coordenada x.

Por otra parte, se hace uso de los registros E y D. El primero de ambos almacena *el signo de la coordenada* y, cuyo valor absoluto está contenido en el registro B.

El registro D almacenará el signo de la coordenada x, cuyo valor absoluto está contenido en el registro C.

El signo (positivo o negativo) se determina con un 01 H para el positivo y un FFH para el negativo. Debo recordar que los números entre 80Hy FFH pueden ser interpretados, como en este caso, como números negativos (complementos a dos).

Es muy importante recordar que el registro doble alternativo H'L' mencionado en el Capítulo cuarto, debe ser salvado antes de acceder a la rutina DRAW (por ejemplo, guardándolo en el STACK), pues su valor será modificado en la propia rutina ROM.

Si el valor de H' L' es modificado y no es restaurado a su valor apropiado, antes de RETornar al BASIC, el programa se autodestruirá al intentarlo.

Un ejemplo de cómo usar DRAW en lenguaje ensamblador es el reflejado en el listado assembler "Prog. 12.5" :

===

Especificaciones : "Prog 12.5" para ZX Spectrum 16K/48K.
Descripciòn General : Diversos ejemplos de lineas DRAW.
Longitud : 67 bytes.
Entrada : Ningùn requerimiento.
Salida : Pixels positivos en diversas zonas de la memoria de pantalla
Registros Usados : B , C , D , E en programa
A , B , C , D , E , H , L en rutina ROM
Nota importante : El registro doble alternativo H'L' se utiliza en la rutina DRAW. Antes de RETornar al BASIC debe contener el valor anterior a la llamada.

```
7000           00100   ORG 7000H       ;
7000      D9   00110   EXX             ; PASA A SET ALTERNATIVO
7001      E5   00120   PUSH HL         ; GUARDA H'L' EN STACK
7002      D9   00130   EXX             ; VUELVE A SET NORMAL
7003    018058         00140   LD BC,22656     ; B=58H=88D  C=80H=128D
7006    CDE522         00150   CALL PLOT       ; PLOT 128,88
7009    013232         00160   LD BC,12850     ; B=32H=50D  C=32H=50D
700C    110101 00170   LD DE,257       ; SGN B = +      SGN C = +
700F    CDBA24         00180   CALL DRAW       ; DRAW 50,50
7012    018058         00190   LD BC,22656     ; B=58H=88D  C=80H=128D
7015    CDE522         00200   CALL PLOT       ; PLOT 128,88
7018    013232         00210   LD BC,12850     ; B=32H=50D  C=32H=50D
701B    11FF01         00220   LD DE,511       ; SGN B= -       SGN C = +
701E    CDBA24         00230   CALL DRAW       ; DRAW 50,-50
7021    018058         00240   LD BC,22656     ; B=58H=88D  C=80H=128D
7024    CDE522         00250   CALL PLOT       ; PLOT 128,88
7027    013232         00260   LD BC,12850     ; B=32H=50D  C=32H=50D
702A    11FFFF         00270   LD DE,65535     ; SGN B = -      SGN C = -
702D    CDBA24         00280   CALL DRAW       ; DRAW -50,-50
7030    018058         00290   LD BC,22656     ; B=58H=88D  C=80H=128D
7033    CDE522         00300   CALL PLOT       ; PLOT 128,88
7036    013232         00310   LD BC,12850     ; B=32H=50D  C=32H=50D
7039    1101FF         00320   LD DE,65281     ; SGN B = +      SGN C = -
703C    CDBA24         00330   CALL DRAW       ; DRAW -50,50
703F      D9   00340   EXX             ; PASA A SET ALTERNATIVO
7040      El   00350   POP HL          ; RECUPERA H'L'
7041      D9   00360   EXX             ; PASA A SET NORMAL
7042      C9   00370   RET             ; VUELVE AL BASIC
0000           00380   END             ;

22E5    PLOT
24BA    DRAW
```

===

El valor original del registro doble alternativo H'L' se guarda en el STACK o pila de máquina gracias a la instrucción PUSH HL. El usuario puede tener acceso al set de registros alternativo y al set de registros normal, pero normalmente a uno de los dos, nunca a los dos a la vez.

Al ejecutar un programa en código máquina, se comienza a trabajar con el set de registros normal (Registros A, B, C, D, E, H, L...).

Si queremos trabajar con el set de registros alternativo (Registros A', B', C', D', E', H', L'...) hay que advertirselo al microprocesador. Para ello se usa entre otros *el mnemónico EXX*. Este mnemónico procesa el cambio de valores entre el set alternativo y el set normal, para los registros B, C, D, E, H y L.

Estos últimos pasarán a contener los valores de B', C', D', E', H' y L " registros del set alternativo.

El registro doble alternativo contiene valores primordiales para la máquina. Estos valores serán necesarios a la hora de procesar una instrucción RET al BASIC. Si faltan se producirá un temido "CRASH". Precisamente para evitar este problema, el contenido del registro H' L' es almacenado en la pila de máquina mediante la instrucción PUSH HL. Para acceder a este valor, debemos procesar antes un intercambio de valores entre el set alternativo y el normal.

Son las líneas 00110 a 00130 las que cumplen este cometido. Esta operación se cierra con otra instrucción EXX, para devolver los valores antiguos al set normal y al alternativo.

El resto del programa, excepto las últimas cuatro instrucciones está estructurado en 4 grupos de 5 instrucciones assembler cada uno, que tienen la función de realizar una línea recta mediante DRAW , siempre en una dirección diferente.

En cada uno de los grupos se define en primer lugar el pixel de partida de la línea a imprimir, que es igual a 128,88. Se hace uso de la rutina ROM PLOT, ubicada a partir de la dirección E522H.

Las coordenadas se cargan en los registros B y C y la propia rutina de la memoria de sólo escritura se encarga de realizar la impresión del pixel.

A continuación se carga en los registros B y C los valores que posteriormente serán utilizados en la rutina DRAW. Yo mismo los he fijado en 50, 50. Es importante fijarse si estos u otros valores serán aceptados como válidos, pues una vez comenzada la rutina, el valor de H'L' será modificado y no restaurado hasta el final. Si aparece un error durante la misma, se retornará al BASIC, sin haber restaurado el valor de H'L ', cosa que podría originar problemas.

Seguidamente se introducen en los registros D y E, el signo de las coordenadas C y B, respectivamente. El valor 01 H interpretará el contenido de C o B como positivo. El valor 0FFH (-1 d) interpretará el contenido de C o B como negativo.

Para pasar a la impresión de la recta, no hay más que "llamar" a la rutina DRAW , ubicada a partir de la dirección 24BAH. Tal y como ocurría en la rutina PLOT, la DRAW se encargará de

realizar lo que de ella exigimos. Una vez acabada la rutina (que en su totalidad ocupa en ROM aproximadamente 100 bytes), se RETornará al BASIC y se procesará la próxima recta.

Los únicos valores que cambian de grupo a grupo son los pertenecientes a los signos de las coordenadas, pues el pixel de partida y los valores absolutos de las coordenadas no experimentan variación alguna.

Una vez son impresas las cuatro rectas, no queda más que retornar al BASIC. Es muy importante volver a recoger el valor para H'L', que está almacenado en el stack. Análogamente al comienzo, debemos primero pasar al set alternativo. Entonces se recupera el preciado valor, guardado en la pila de máquina desde el comienzo. Se utiliza la instrucción complementaria a PUSH, es decir, POP. POP HL introduce en HL el contenido primitivo de H'L'.

Tal y como se hizo al principio hay que volver a intercambiar los valores entre el set alternativo y el normal. De esta manera H'L' recupera definitivamente su valor, y se puede procesar, sin peligro, la instrucción de RETorno. Este es el programa BASIC que carga los códigos del programa assembler anterior:

```
 10 REM PROG 12.5
 20 CLEAR 28650: LET T=0
 30 FOR F=28672 TO 28738
 40 READ A: POKE F,A
 50 LET T=T+A: NEXT F
 60 IF T< >7467 THEN PRINT "ERROR EN DATAS" : STOP
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA "
 80 PAUSE 0: RANDOMIZE USR 28672
 90 DATA 217 , 229 , 217
100 DATA 1 , 128 , 88 , 205 , 229 , 34
110 DATA 1 , 50 , 50 , 17, 1 , 1
120 DATA 205 , 186 , 36
130 DATA 1 , 128 , 88 , 205 , 229 , 34
140 DATA 1 , 50 , 50 , 17 , 255 , 1
150 DATA 205 , 186 , 36
160 DATA 1 , 128 , 88 , 205 , 229 , 34
170 DATA 1 , 50 , 50 , 17 , 255 , 255
180 DATA 205 , 186 , 36
190 DATA 1 , 128 , 88 , 205 , 229 , 34
200 DATA 1 , 50 , 50 , 17 , 1 , 255
210 DATA 205 , 186 , 36
220 DATA 217 , 225 , 217
230 DATA 201
```

El programa assembler "Prog. 12.5" equivaldría al BASIC:

```
10 PLOT 128,88
20 DRAW 50,50
```

```
30 PLOT 128,88
40 DRAW 50,-50
50 PLOT 128,88
60 DRAW -50,-50
70 PLOT 128,88
80 DRAW 50,-50
```

La diferencia en la velocidad de ejecución entre la orden DRAW en BASIC y en código máquina es realmente notable. Las dos usan la misma base, una rutina en código máquina en ROM, pero la primera debe ser traducida antes de poder ser procesada, mientras la segunda es entendida automáticamente por el microprocesador. DRAW, al igual que PLOT, puede hacer uso de las funciones OVER o INVERSE. Para ello habría que modificar la variable correspondiente (P-FLAG).

El programa assembler "Prog 12.5" puede ser mejorado, haciéndolo más corto. Observa que son muchas las instrucciones exactamente iguales que se repiten. Prueba a acortarlo, disponiendo una subrutina para aquellas ordenes reiterativas, antes de pasar al próximo programa.

El programa "Kalidex"

He querido realizar un programa más completo, en el que entren no solamente las órdenes PLOT y DRAW, sino también otras, muy usuales en programación assembler. Este programa realiza un gráfico simétrico en alta resolución, consecuencia del uso de DRAW y la función OVER 1.

Antes de pasar a comentarlo, quiero mostraros su equivalente BASIC, y os propongo que, sin mirar la solución final, intentéis programar lo que sería en código máquina.

Comienza analizando el problema y sigue cada pensamiento paso a paso. Un consejo: una vez hayas hecho algo, grábalo inmediatamente, antes de probarlo. Un fallo podría originar un CRASH y tendrías que comenzar otra vez de cero.

Equivalente BASIC a "Kalidex":

```
  5 CLS
 10 FOR F=0 TO 87
 20 PLOT 128, 88
 30 DRAW 50, F
 40 PLOT 128, 88
 50 DRAW -50, F
 60 PLOT 128, 88
 70 DRAW 50, -F
 80 PLOT 128, 88
 90 DRAW -F, -50
100 PLOT 128, 88
110 DRAW F, 50
```

```
120 PLOT 128, 88
130 DRAW -F, 50
140 PLOT 128, 88
150 DRAW F, -50
160 PLOT 128,88
170 DRAW -F, -50
180 NEXT F
190 OVER l:GOTO 10
```

Bueno, si te has dado por vencido -cosa que reprocho-, o si quieres comparar resultados positivos -eso me gusta más-, tienes permiso para seguir leyendo.

Dirígete al listado assembler y fíjate en los mnemónicos.

La primera instrucción tiene la función de borrar la pantalla. Se procesa una llamada a la rutina de la memoria ROM de CLS.

Las dos instrucciones siguientes abren el canal dos y realizan un CALL a la subrutina OPEN. De esta manera, el canal de pantalla es abierto.

Estas tres instrucciones son un tanto tradicionales y aparecen en primer lugar de los programas.

En "Kalidex" se hace uso de la rutina ROM DRAW, y una cosa que no debéis olvidar nunca si la utilizáis, es que el registro doble alternativo H'L' hay que salvarlo antes de acceder a ella.

El sistema usado aquí es el mismo que el usado en listado assembler anterior. Primero hay que acceder al set de registros alternativo. EXX intercambia los valores entre los registros B a L y B' a L'. En el STACK se almacena el valor de HL, que en estos momentos contiene el valor de H'L'. Para acceder de nuevo a los valores del set de registros normal se procesa otro EXX.

La función OVER e INVERSE puede ser usada, al igual que con PLOT, con DRAW. Como ya comentamos, es la variable P-FLAG la que almacena los valores de OVER e INVERSE. Si los dos primeros bits de dicha variable tienen el valor 0, se activa OVER 0. Por el contrario, si contienen ambos el valor 1, la función OVER 1 entra en juego. Se puede acceder a P-FLAG mediante el registro doble IY. La distancia entre P-F LAG e IY es de 87 bytes. El sistema usado es el direccionamiento indexado, tal y como en el último ejemplo de PLOT. Se introduce el valor 0, de manera que los bits 0 y 1 (y también el resto, aunque esto no nos importa) tengan el valor 0 y se acceda a la función OVER 0.

Seguidamente se introduce en el registro BC el valor 0032 H. El registro C almacena la coordenada "constante" de la orden DRAW, es decir 32H o 50d. El otro registro almacena la coordenada "variable", cuyo valor inicial será de 0, al igual que en el bucle. Al introducir directamente en el registro BC, nos ahorramos un byte, comparado con el caso que cargáramos los valores en B y C separadamente.

Hay que contar con que el registro BC va a ser usado, irremediablemente, pocas instrucciones más adelante, y para no perder este valor, es necesario almacenarlo en alguna parte. El STACR nos viene aquí de maravilla. La instrucción PUSH BC guarda en la pila de máquina el valor que en ese momento contenía BC, precisamente 0032H.

Si queremos ejecutar una orden DRAW, tenemos que advertir antes al microprocesador el pixel de referencia. En caso contrario, éste será el de coordenadas (0/0). Nuestro pixel de referencia será el de coordenadas (128/88), exactamente el centro de la pantalla, en el formato de alta resolución. Para ello no tenemos más que PLOTear este punto determinado, con la rutina PLOT , que ya conocemos. Es aquí donde volvemos a usar el registro doble BC. Lo necesitamos para definir las coordenadas del punto. Si no hubiéramos almacenado aquel valor 0032H en el STACK, ahora lo perderíamos al intentar ejecutar el PLOT.

Este pixel va a ser, durante todo el programa, el pixel de referencia de los DRAW . Antes de cada uno de ellos, hay que "recordárselo" de nuevo a la máquina. Este proceso repetitivo que sería la ejecución del PLOT 128,88 puede ser almacenado en una subrutina, y acceder a ella antes de cada DRAW. Recibe el nombre de PLOT1, y su punto de entrada es la dirección de memoria 7074H. En esta subrutina cargamos en el registro doble BC el valor correspondicnte al pixel (128/88), y se llama a la rutina ROM PLOT. De esta manera se imprime el pixel en el lugar que queremos. El registro BC debe volver de la subrutina con el valor para la siguiente orden DRAW. Recordarás que lo almacenamos al principiv en el STACK. Quizá pienses que la instrucción POP será suficiente para resolvernos el problema, pero esto no es así.

Recuerda que si el microprocesador ejecuta una orden CALL, debe "acordarse" de la dirección de memoria a la que el RET le retornará. En BASIC, si se procesa un GO SUB, el Spectrum debe "acordarse" de la línea a la que el RETURN le devolverá. En el caso de la orden BASIC GO SUB, los números de línea se almacenan en la pila del GO SUB, zona de la memoria exclusivamente dedicada a este efecto. Cuando el Spectrum se encuentra el comando RETURN, él recupera el valor de la pila del GO SUB, y se retorna a la línea y al número de orden correspondiente.

Cuando un CALL se ejecuta, la dirección de memoria a la que se debe retornar, se almacena en el STACK. Cuando el microprocesador topa con un RET, se recupera este valor, y el Z80 sabe volver a la dirección adecuada.

Si queremos recuperar el registro BC, almacenando anteriormente en la pila de máquina, dentro de la misma subrutina con un POP BC, introduciremos en este registro la dirección de memoria a la que se debe retornar, y no el valor que queremos nosotros.

Esto se debe a que el primer valor que tras un CALL se encuentra en el STACK es precisamente el que necesita la máquina para retornar. ¡Hay que hacer algo! Debemos fijarnos que el valor que queremos recuperar e introducir en BC se encuentra tras el almacenado por la misma máquina.

La solución más sencilla es almacenar en cualquier lugar la dirección de retorno al principio de la subrutina. De esta manera, guardaremos este valor, que será necesario a la hora de procesar el RET, pues sino, el Z80 no sabría a qué dirección de memoria debería retornar. Al final de la

subrutina habría que volver a introducir en la pila de máquina la dirección de retorno. para que no hubiera problema. Así, se podría utilizar como quisiéramos los PUSH y los POP dentro de la subrutina, sin peligro de CRASH . Existen diferentes métodos para acceder a la dirección de retorno. Lo primero será recuperarlo del STACK mediante un POP. Una vez se encuentre en un registro doble, podemos guardarlo allí hasta el final de la subrutina. Entonces deberíamos hacer un PUSH, para volver a introducir el valor de retorno en el lugar inicial y adecuado.

Sin embargo, aquí nos encontramos con el problema que la rutina PLOT, necesaria para fijar el pixel de referencia, va a modificar los valores de todos registros. Este método no sería adecuado a nuestro caso.

Otra posibilidad es almacenar este preciado valor en una dirección de la RAM, y recuperarlo al final. El sistema sería recuperar el valor mediante POP, y después, por medio de direccionamiento extendido, introducirlo en una dirección de memoria. Un buen sitio para almacenar el valor es en las variables del sistema, en la dirección 5CB0H (23728 d) , pues ni ésta ni la siguiente son usadas por el microprocesador. Son dos "bytes de reserva" cuyo contenido no puede ser modificado por ninguna causa -a no ser que se desenchufe, o que se vuelvan a introducir otros valores-.

En nuestra subrutina, la dirección de retorno se recupera con POP HL. El valor se introduce en la dirección 5CB0H, donde estará seguro.

Ya podemos seguir con la subrutina. Seguidamente se fija en el registro BC el pixel, que será impreso por la rutina PLOT.

El valor con el que debe retornar BC es el que más tarde se usará en la rutina DRAW . Este mismo se inicializó al principio y se almacenó en el STACK. Ahora no habrá peligro de que un POP BC nos introduzca la dirección de retorno, pues ésta ya la introdujimos en la dirección 5CB0H.

El POP BC recuperará el valor introducido inicialmente. Debemos pensar que este mismo va a ser utilizado más adelante, cuando se vuelva a acceder a la rutina PLOT1. Esta es la razón de la importancia del PUSH BC, instrucción que sigue inmediatamente al POP. Ambas no afectan ni a otros registros ni a ningún indicador, solamente modifican el de BC.

Antes de querer retornar es necesario volver a introducir en la pila de máquina la dirección de retorno, que ahora ocupa la dirección 5CB0H. El método es precisamente el contrario que al principio de la subrutina. Primero se carga en un registro doble, que no sea BC, el contenido de la dirección 5CB0H. Este registro, puede ser, por ejemplo, el HL. Para introducir el valor de retorno de nuevo en el STACK se usa un simple PUSH HL.

Ahora está todo en orden: BC contiene las coordenadas del DRAW , en la pantalla aparece un pixel con coordenadas (128/88), y la instrucción de RETorno ha funcionado perfectamente. La subrutina ha cumplido su función.

Ahora nos encontramos en la línea 00210 del listado. El valor 257 d se introduce en el registro doble DE, lo que equivale a introducir en D y E el valor 1. Al acceder a la rutina ROM DRAW ,

el registro BC contiene las coordenadas x, y y el DE los signos de estas coordenadas. En este caso, B y C se interpretarán como positivos.

Ya tenemos todos los datos necesarios para poder ejecutar el DRAW sin ningún peligro. El punto de entrada de la rutina es la dirección 24BA H/9402 d. Con los datos que disponemos ahora, una entrada en esta rutina equivale a la orden BASIC DRAW 50, F, donde F es igual a 0.

Desde la llamada a la subrutina PLOT1 hasta la línea 00310, el programa se puede estructurar de la siguiente manera: 3 grupos de 3 instrucciones cada uno. La primera instrucción es un CALL PLOT1, que tiene la función de fijar el pixels de referencia y traspasar el valor adecuado de la rutina DRAW al registro doble BC. La segunda introduce en DE los signos de las coordenadas. Empezaron siendo ambas positivas, para cambiar luego a positiva-negativa, negativa-positiva y negativa-negativa, en este orden. Ejercita y explicate a tí mismo, cómo van progresando estos signos. La última instrucción de estos 3 grupos es un CALL a la rutina DRAW. Al llegar a este punto, se tiene la certeza que todos los valores están situados en los registros correspondientes, y que no tiene por qué ocurrir ningún fallo.

La estructura de "Kalidex" hasta la línea 00430, es la siguiente: Son 4 grupos de 3 instrucciones cada uno. Difiere solamente en una instrucción, comparado con los grupos anteriores. Precisamente es la primera. No se llama a la subrutina PLOT1 directamente, sino a otra, denominada PLOT2, cuyo punto de entrada es la dirección 7085H (28805 d). Recuerda que en el programa BASIC, primero iba la coordenada variable (F) en segundo lugar, determinando la coordenada de altura (y). Más tarde aparecían 4 instrucciones DRAW, en las que esta variable F aparecía en primera posición, determinando la coordenada x. También fue así en "Kalidex": la rutina PLOT1 carga en el registro B (coordenada y) el valor variable (al inicio = 0) . El registro C (coordenada x) contiene el valor fijo (=32H 50d) .

La subrutina PLOT2 carga en el registro B el valor fijo, y almacena en el C el variable, justamente lo contrario que P LOT1. Analicemos esta subrutina: se va a hacer uso de la PLOT1 , para fijar el pixel de referencia en las coordenadas (128/88) , y para introducir en BC los valores del DRAW . Una vez haya retornado a PLOT2, no hay más que intercambiar los valores entre B y C. De esta manera, C contiene el valor que antes estaba en B, el variable. El registro B va a contener el fijo, el valor 32H.

Como esta subrutina se "ayuda" de la PLOT1 , en la que aparece un PUSH y un POP, el valor de retorno, guardado en el STACK, debe ser almacenado en cualquier otra parte. Para ello he elegido una dirección de memoria, la 6FFEH (28670 d). Esta es bastante segura, pues se encuentra sobre el RAMTOP y bajo el programa. El sistema usado es el mismo que en PLOT1 : Un POP que toma el valor de retorno, el cual es almacenado en una dirección de memoria. Esto se realiza al principio de la subrutina. Al final de la misma, se vuelve a recuperar el valor, para introducirlo de nuevo en la pila de máquina. La instrucción RET nos devolverá al programa principal.

Como dije, las otras dos instrucciones de los 4 grupos, son iguales a las dos instrucciones Últimas del grupo primero.

Se trata de introducir en DE los signos de las coordenadas. Estos van cambiando de positivo-positivo, positivo-negativo, negativo-positivo y negativo-negativo. La última instrucción realiza un CALL a la subrutina DRAW de la memoria ROM.

El primer conjunto de 4 grupos equivale a las sentencias 20 a 90 del programa BASIC anterior. La segunda coordenada es una variable, mientras la primera no varía.

El segundo conjunto de 4 grupos de instrucciones equ ivale a las sentencias 100 a 170. La primera coordenada es variable, y la segunda mantiene siempre igual a 32H .

Ya están terminados los DRAW, ahora hay que incrementar la variable, comprobar si ha terminado y tomar en cada caso la decisión oportuna. Primeramente se recupera B, almacenado en el STACK. B contiene la variable, cuyo valor inicial es 0. Se podría comparar a F en el programa BASIC.

He querido darte la opción de salir del programa. Una orden RET se ejecutará siempre que esté pulsada la tecla R.

El valor de B es INCrementado. De esta manera, los DRAW siguientes serán diferentes, pues las coordenadas no son las mismas. El límite máximo de B es 57H (87d), pues un valor mayor produciría error tipo "B-lnteger out of range". Este límite se introduce en el registro A, y con él se compara el registro B.

Si B es igual a 57H, el indicador de cero tomará el valor 1. En cualquier otro caso, no. Si el indicador de cero es igual a 0 (B < > 57 H), el programa ejecuta un salto relativo hacia la sentencia, lo que equivale a procesar otros DRAW. esta vez con otras coordenadas.

Esta parte del listado puede compararse con el comando NEXT. NEXT incrementa la variable, y ejecuta de nuevo el bucle si no se ha alcanzado el límite máximo, definido en el FOR.

En caso de que el indicador de cero sea igual a 1, no se ejecuta el salto relativo. La siguiente instrucción modifica el contenido de P-FLAG, accediendo a la función OVER 1.

Un salto relativo envía al programa al principio, comenzando de nuevo el bucle.

Este proceso se repetirá hasta que se pulse la tecla R. En este caso, se accede a una rutina denominada RET. En esta rutina se restaura el valor de H'L', condición indispensable para un retorno al BASIC seguro. El programa ha acabado. La orden RET le devolverá al BASIC.

Teniendo ya un ejemplo de uso de DRAW, el prototipo "Kalidex", prueba tú mismo con otras fórmulas que, como ésta, realicen dibujos en alta realicen dibujos en alta resolución.

Listado Assembler "Kalidex" :
===
Especificaciones : "Kalidex" para ZX Spectrum 16K/48K
Descripciòn General : Ejemplo de un dibujo en alta resoluciòn.
Longitud : 148 bytes.
Entrada : Ningùn requerimiento.
Salida : Memoria de pantalla ocupada en parte por un dibujo
simètrico

Registros Usados : A , B , C , D , E , H , L
Nota importante : El registro doble alternativo H'L' debe ser guardado al principio de la rutina y restaurado al final.

```
7000             00100        ORG 7000H        ;
7000   CD6B0D    00110        CALL CLS         ; BORRA PANTALLA
7003   33E02     00120        LD A,2D          ; ABRE
7005   CD0116    00130        CALL OPEN        ; CANAL N,  DOS
7008   D9        00140        EXX              ; SET ALTERNATIVO
7009   E5        00150        PUSH HL          ; GUARDA HL
700A   D9        00160        EXX              ; SER NORMAL
700B   FD365700  00170        LD (IY+87),0     ; ACCESO A OVER 0
700F   013200    00180  COMI  LD BC,50         ; B = 0 C = 50
7012   C5        00190  OTRO  PUSH BC          ; ALMACENA VARIABLE B
7013   CD7470    00200        CALL PLOT1       ;
7016   110101    00210        LD DE,257        ; SGB B=+     SGN C=+
7019   CDBA24    00220        CALL DRAW        ; DRAW 50,F
701C   CD7470    00230        CALL PLOT1       ;
701F   1101FF    00240        LD DE,65281      ; SGN B=+     SGN C=-
7022   CDBA24    00250        CALL DRAW        ; DRAW -50,F
7025   CD7470    00260        CALL PLOT1       ;
7028   11FF01    00270        LD DE,511        ; SGN B=-     SGN C=+
702B   CDBA24    00280        CALL DRAW        ; DRAW 50,-F
702E   CD7470    00290        CALL PLOTl       ;
7031   11FFFF    00300        LD DE,65535      ; SGN B=-     SGN C=-
7034   CDBA24    00310        CALL DRAW        ; DRAW -50,-F
7037   CD8570    00320        CALL PLOT2       ;
703A   110101    00330        LD DE,257        ; SGN B=+     SGN C=+
703D   CDBA24    00340        CALL DRAW        ; DRAW F,50
7040   CD8570    00350        CALL PLOT2       ;
7043   11FF01    00360        LD DE,65281      ; SGN B=+     SGN C=-
7046   CDBA24    00370        CALL DRAW        ; DRAW -F,50
7049   CD8570    00380        CALL PLOT2       ;
704C   1101FF    00390        LD DE,511        ; SGN B=-     SGN C=+
704F   CDBA24    00400        CALL DRAW        ; DRAW F,-50
7052   CD8570    00410        CALL PLOT2       ;
7055   11FFFF    00420        LD DE,65535      ; SGN B=-     SGN C=-
7058   CDBA24    00430        CALL DRAW        ; DRAW -F,-50
705B   C1        00440        POP BC           ; RECUPERA VARIABLE B
705C   3EFB      00450        LD A,251         ; SEMIFILA Q-T
705E   DBFE      00460        IN A, (254)      ; LEE TECLADO
7060   E608      00470        AND 8            ; TECLA "R" PULSADA?
7062   280C      00480        JR Z,RET         ; SI ES ASI,RET
7064   04        00490        INC B            ; B=B+1
7065   3E57      00500        LD A,87          ; 87=LIMITE MAXIMO
7067   B8        00510        CP B             ; COMPARACION
7068   20A8      00520        JR NZ,OTRO       ; SI B< >87, OTRO
```

```
706A    FD365703    00530           LD (IY+87) ,3     ; ACCESO A OVER 1
706E      189F      00540           JR COMI           ; VE AL COMIENZO
7070    D9      00550       RET EXX       ; SET ALTERNATIVO
7371    El      00560           POP HL        ; RECUPERA HL
7072    D9      00570           EXX           ; SET NORMAL
7073    C9      00580           RET           ; RETORNO A BASIC
7074    El      00590   PLOTl   POP HL        ; COGE DIREC.RETORNO
7075      22B05C    00600           LD (23728),HL     ; ALMACENALA EN 5CB0H
7078      018058    00610           LD BC,22656       ; B=58H=88D  C=80H=128D
707B      CDE522    00620           CALL PLOT         ; PLOT 128,88
707E    Cl      00630           POP BC        ; COGE VALOR PARA DRAW
707F    C5      00640           PUSH BC       ; ALMACENALO
7080      2AB05C    00650           LD HL, (23728)    ; RECUPERA DIR.RETORNO
7083    E5      00660           PUSH HL       ; ALMACENALA EN STACK
7084    C9      00670           RET           ;
7085    El      00680   PLOT2   POP HL        ; COGE DIREC.RETORNO
7086      22FE6F    00690           LD (28670) ,HL    ; ALMACENALA EN 6FFEH
7089      CD7470    00700           CALL PLOTl        ;
708C    48      00710           LD C,B        ; PASA VARIABLE A C
708D      0632 00720            LD B,50       ;
708F      2AFE6F    00730           LD HL, (28670)    ; RECUPERA DIR.RETORNO
7092    E5      00740           PUSH HL       ; ALMACENALA EN STACK
7093    C9      00750           RET           ;
0000            00760           END           ;

22E5    PLOT
24BA    DRAW
700F    COMI
7012    OTRO
7070    RET
7074    PLOT1
7085    PLOT2
```

===

Y prepararos para introducir el listado BASIC de "Kalidex":

```
10 REM *** KALIDEX ***
20 REM Juan Mtz.Velarde,1984
30 CLEAR 28650: LET T=0
40 FOR F=28672 TO 28819
50 READ A: POKE F,A
60 LET T=T+A: NEXT F
70 IF T<>18664 THEN PRINT "ERROR EN DATAS": STOP
80 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
   MAQUINA": PAUSE 0
```

```
 90 RANDOMIZE USR 28672
100 DATA 205 , 107 , 13 , 62 , 2 , 205 , 1 , 22 , 217 , 229 , 217 , 253 , 54 , 87 , 0
110 DATA 1 , 50 , 0 , 197 , 205 , 116 , 112 , 17 , 1 , 1 , 205 , 186 , 36 , 205 , 116 ,
    112 , 17 , 1 , 255 , 205 , 186 , 36 , 205 , 116 ,112 , 17 , 255 , 1 , 205 , 186 ,
    36 , 205, 116 , 112 , 17 , 255 , 255 , 205 , 186 , 36
120 DATA 205 , 133 , 112 , 17 , 1 , 1 , 205 , 186 , 36 , 205 , 133 , 112 , 17 , 1 , 255 ,
    205 , 186 , 36 , 205 , 133 , 112 ,17, 255 , 1 , 205 , 186 , 36 , 205 , 133 , 112 ,
    17 , 255 , 255 , 205 , 186 , 36
130 DATA 193 , 62 , 251 , 219 , 254 , 230 , 8 , 40 , 12 , 4 , 62 , 87 , 184 , 32 , 168 ,
    253 , 54 , 87 , 3 , 24 , 159
140 DATA 217 , 225 , 217 , 201
150 DATA 225 , 34 , 176 , 92 , 1 , 128 , 88 , 205 , 229 , 34 , 193 , 197 , 42 , 176 ,
    92 , 229 , 201
160 DATA 225 , 34 , 254 , 111 , 205 , 116 , 112 , 72 , 6 , 50 , 42 , 254 , 111 , 229 ,
    201
```

3. EL COMANDO "CIRCLE"

Este es el último comando gráfico de importancia que nos queda por explicar. En lenguaje BASIC tiene la forma x, y, r. Los dos primeros parámetros definen las coordenadas y el tercero el radio de la circunferencia a dibujar. CIRCLE usa la pantalla de alta resolución.

El círculo mayor que podemos dibujar es el de coordenadas 128, 88 y radio 87. En cualquier caso que nos excedamos, aparecerá el informe "B-lnteger out of range".

Dentro de la memoria ROM se encuentra el algoritmo. un complicado y largo sistema para dibujar la circunferencia. Este algoritmo hace uso de las rutinas PLOT y DRAW , tratadas anteriormente. Esta es la causa de la lentitud de una orden de este tipo. Incluso en código máquina parece lenta.

El comando CIRCLE es muy útil, a la hora de realizar cualquier dibujo en alta resolución.

Si siguieramos la línea de pensamiento usada en las anteriores órdenes gráficas, aquí también sería posible ejecutar CIRCLE si determinados registros contienen determinados valores.

Lamentablemente, ésta es la excepción que confirma la regla.

No existe posibilidad alguna de éxito si seguimos por este camino. Si analizáramos el algoritmo del CIRCLE, veríamos que usa el STACK o PILA DEL CALCULADOR. Esta es una zona de la memoria RAM reservada especialmente al uso del CALCULADOR. Este es un dispositivo interno de la máquina, que permite hacer las cuatro operaciones aritméticas, las trigonométricas, y otras.

El funcionamiento del calculador es muy complejo y realmente se aparta de nuestro camino.

Solamente necesitamos saber que la rutina CIRCLE almacena y utiliza determinados valores que se guardan en la pila del calculador. Estos valores son procesados más adelante para realizar el dibujo del círculo.

La única posibilidad que tenemos para dibujar circunferencias en código máquina es almacenar en el STACK del calculador una serie de valores y acceder a la rutina de CIRCLE en una dirección muy concreta.

Es muy importante saber que el registro doble alternativo H'L' es usado por la rutina CIRCLE. Al igual que pasó con DRAW , tenemos que guardar el valor del registro y restaurarlo antes de retornar al BASIC.

Una rutina de la ROM, muy especial, llamada STACK-A (Guarda A) almacena en la pila del calculador el contenido del registro A. Esta rutina se ubica a partir de la dirección de memoria 2D28H ó 11560 d. Utilizaremos esta rutina tan útil para guardar los valores apropiados.

La dirección exacta de la rutina CIRCLE por donde debemos acceder es la 232D H o 9005 d. Esta rutina necesita tener en la pila del calculador los siguientes valores.

1. El primero de la pila debe ser el radio del círculo a dibujar.

2. A éste le seguirá la coordenada y (0-175) del mismo círculo.

3. En último lugar se debe encontrar la coordenada x (0-255) de la circunferencia.

La pila del calculador tiene la misma estructura que la pila de máquina. Es una caja en la que podemos meter un número determinado de elementos. Pero siempre que queramos sacer uno, ha de ser el de más arriba. Si intentamos sacar el de abajo del todo, sin sacar antes los que se encuentran sobre él, la caja se romperá. También si queremos sacar algo de la caja, pero está vacía, se romperá.

De esta manera, si el primero de la pila tiene que ser el radio del círculo, éste será el último en ser introducido.

El primero que hay que meter en la caja es la coordenada x, pues ésta será la última en ser sacada. Después irá la coordenada y, y tras ésta, el radio.

Es vital seguir este orden, pues así interpretará el microprocesador los números del STACK.

Este programa es un ejemplo de cómo usar esta orden:

Listado assembler "Prog 12.6"
===
Especificaciones "Prog 12.6" para ZX Spectrum 16K/48K.
Descripciòn General : CIRCLE 128 , 88 , 87 en còdigo màquina.
Longitud : 33 bytes.

Entrada : Ningùn requerimiento.
Salida : Circunferencia en memoria de pantalla.
Registros Usados : A en programa
A , B , C , D , E , H , L en rutina ROM
NOTA IMPORTANTE : El registro doble alternativo HL debe ser guardado antes de la llamada a la rutina CIRCLE,y restaurado antes del RETorno al BASIC.

```
7000              00100   ORG 7000H       ;
7000   CD6B0D     00110   CALL CLS        ; BORRA PANTALLA
7003   3E02       00120   LD A,2 DN       ;   ABRE
7005   CD0116     00130   CALL OPEN       ; CANAL DOS
7008   D9         00140   EXX             ; SET ALTERNATIVO
7009   E5         00150   PUSH HL         ; GUARDA H'L'
700A   D9         00160   EXX             ; SET NORMAL
700B   3E80       00170   LD A,128        ; COORDENADA X
700D   CD282D     00180   CALL STACK-A    ; ALMACENALA EN STACK-CALC
7010   3E58       00190   LD A,88         ; COORDENADA Y
7012   CD282D     00200   CALL STACK-A    ; ALMACENALA EN STACK-CALC
7015   3E57       00210   LD A,87         ; RADIO
7017   CD282D     00220   CALL STACK-A    ; ALMACENALO EN STACK-CALC
701A   CD2D23     00230   CALL CIRCLE     ; CIRCLE 128 , 88 , 87
701D   D9         00240   EXX             ; SET ALTERNATIVO
701E   El         00250   POP HL          ; RECUPERA H'L'
701F   D9         00260   EXX             ; SET NORMAL
7020   C9         00270   RET             ;
0000              00280   END             ;

0D6B   CLS
1601   OPEN
232D   CIRCLE
2D28   STACK-A
```
===

Este programa equivale a la orden BASIC CIRCLE 128, 88, 87. Es el circulo de mayores dimensiones que se puede dibujar en la pantalla.

En el programa, primero se borra la pantalla mediante la rutina CLS y se abre a continuación el canal dos.

Más adelante se guarda en la pila de máquina el valor H' L' que como ya sabemos, será modificado en la rutina CIRCLE, y nos será indispensable para el RETorno al BASIC.

Mediante la subrutina STACK-A, se van almacenando en la pila de calculador, los valores necesarios para el círculo. En primer lugar, la coordenada x, pues esta será necesitada en último lugar. "Encima" de la coordenada x, debe ir la coordenada y.

El valor del tope de la pila (el primer elemento de la "caja"), será el radio del círculo.

Los valores correspondientes se van introduciendo en el registro A. La subrutina que se usa a continuación (STACK-A) , se ocupa de introducirlos en la pila de máquina.

Una vez se halla llegado a la línea 00230, los tres valores se encuentran en la pila, y no hay peligro para acceder a la rutina de CIRCLE.

Nótese que esta rutina, incluso en código máquina, es bastante lenta. El algoritmo en memoría es largo. Los pixels a calcular son muchos.

Quizá, si estás pensando en realizar círculos en código máquina a velocidad alta, la solución sea almacenar los pixels en un DATA, irlos leyendo uno a uno e imprimirlos mediante la rutina ROM PLOT.

Si se modifican los valores de las variables del sistema que almacenan atributos y las funciones OVER e INVERSE, éstas afectarán también en la ejecución de CIRCLE.

Para abandonar el programa CIRCLE, es necesario restaurar el registro doble alternativo H' L ', a su valor original. Dado que éste está almacenado en el STACK, no tenemos más que recuperarlo con un POP. A continuación, aparece el listado BASIC correspondiente al programa assembler anterior:

```
 10 REM PROG 12, 6
 20 CLEAR 28650: LET T=0
 30 FOR F=28672 TO 28704
 40 READ A: POKE F, A
 50 LET T=T+A: NEXT F
 60 IF T<>3784 THEN PRINT "ERROR EN DATAS" : STOP
 70 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA " : PAUSE 0
 80 RANDOMIZE USR 28672
100 DATA 205 , 107 , 13 , 62 , 2 , 205 , 1 , 22 , 217 , 229 , 217
110 DATA 62 , 128 , 205 , 40 , 45 , 62 , 88 , 205 , 40 , 45 , 62 , 87 , 205 , 40 , 45 ,
    205 , 45 , 35
130 DATA 217 , 225 , 217 , 201
```

El programa "CIRCULOS"

Al igual que "Kalídex" fue un ejemplo del uso de DRAW, "CIRCULOS" lo es de CIRCLE. La versión BASIC es la siguiente:

```
10 FOR F=0 TO 87
20 CIRCLE F, F, F
30 NEXT F
```

La variable F es utilizada como coordenada x, y y como radio.

Este programa crea una bonita imagen de círculos, cada vez más grandes, avanzando por la pantalla. El valor máximo es 87, pues uno superior a éste originaría un informe tipo "B-lnteger out of range". Antes de pasar al listado assembler, intenta programarlo tú mismo en código máquina.

Listado assembler "CIRCULOS" :
===
Especificaciones : "CIRCULOS" para ZX Spectrum 16K/48K.
Descripciòn General : Ejemplo del uso de CIRCLE en un bucle.
Longitud : 33 bytes.
Entrada : Ningùn requerimiento.
Salida : Circunferencias en memoria de pantalla.
Registros Usados : A , B en programa
A , B , C , D , E , H , L en rutina ROM
NOTA IMPORTANTE : El registro doble alternativo HL debe ser guardado antes de la llamada a la rutina CIRCLE,y restaurado antes del RETorno al BASIC.

```
7000                  ORG 7000H       ;
7000  CD6B0D                CALL CLS        ; BORRA PANTALLA
7003  3E02            LD A,2 D        ;   ABRE
7005  CD0116                CALL OPEN       ; CANAL DOS
7008  D9              EXX             ; SET ALTERNATIVO
7009  E5              PUSH HL         ; GUARDA H'L'
700A  D9              EXX             ; SET NORMAL
700B  0600            LD B,0          ; VARIABLE, INICIO = 0
700D  C5        OTRO  PUSH BC         ; ALMACENA VARIABLE
700E  78              LD A,B          ; PASALA A ACUMULADOR
700F  CD282D                CALL STACK-A ; ALMACENA COORDENADA X
7012  C1              POP BC          ; RECUPERA VARIABLE
7013  78              LD A,B          ; PASALA A ACUMULADOR
7014  C5              PUSH BC         ; ALMACENA VARIABLE
7015  CD282D                CALL STACK-A ; ALMACENA COORDENADA Y
7018  C1              POP BC          ; RECUPERA VARIABLE
7019  78              LD A,B          ; PASALA A ACUMULADOR
701A  C5              PUSH BC         ; ALMACENA VARIABLE
701B  CD282D                CALL STACK-A  ; ALMACENA RADIO
701E  CD2D23                CALL CIRCLE     ; CIRCLE B, B, B
7021  C1              POP BC          ; RECUPERA VARIABLE
7022  3EFB            LD A,251        ; SEMIFILA Q-T
7024  DBFE            IN A, ( 254)    ; LEE TECLADO
7026  E608            AND 8           ; TECLA "R" PULSADA ?
7028  2806            JR Z,RET        ;
702A  04              INC B           ; B=B+1
702B  3E57            LD A,87         ; 87 = VALOR MAXIMO
```

```
702D    B8      00370           CP B            ; COMPARA A-B
702E      20DD          00380           JR NZ,OTRO      ;
7030    D9      00390   RET     EXX             ; SET ALTERNATIVO
7031    El      00400           POP HL          ; RECUPERA H'L'
7032    D9      00410           EXX             ; SET NORMAL
7033    C9      00420           RET             ;
0000            00430           END             ;
0D6B    CLS
1601    OPEN
232D    CIRCLE
2D28    STACK-A
700D    OTRO
7030    RET
=======================================================================
```

El programa tiene la siguiente estructura:

Las primeras líneas deben ser ya unas grandes conocidas para nosotros. Se trata de borrar la pantalla, abrir el canal dos y poner a salvo el registro H' L '.

El registro B se utiliza como registro-variable. Su valor se incrementa cada vez que el proceso se repite. El contenido inicial de B es 0. Observa que este registro debe ser almacenado, pues su valor será modificado en la rutina STACK-A.

Para el funcionamiento del CIRCLE, se necesitan 3 valores en la pila del calculador. Estos serán igual al contenido de B.

El valor de B se traspasa a A, y la rutina STACK-A se encarga de almacenarlo en la pila del calculador.

Antes y después de traspasar el contenido de B al acumulador, éste se recupera y se almacena en el STACK, para que no se pierda.

Una vez se repita esto 3 veces, se accede a la rutina CIRCLE. Esta dibujará la circunferencia en la pantalla.

El valor de B será comparado más adelante, por eso se recupera del STACK.

Seguidamente aparecen dos grupos de instrucciones. El primero (líneas 00310 a 00340) comprueba si la tecla "R" está pulsada. En ese caso, se RETornará inmediatamente al BASIC, restaurando antes la variable H' L '.

El segundo grupo de instrucciones (líneas 00350 a 00380) tienen la función de INCrementar el registro-variable, y de comprobar si no excede el límite. Comprueba este apartado con el de "Kalidex". El proceso se repetirá hasta que B alcance el valor 87d, en los límites de la pantalla.

Como dato significativo del "CIRCULOS", la velocidad de la versión BASIC es de aprox. 1:09 seg. En código máquina tarda poco menos: 1:07 seg. Los códigos del programa "CIRCULOS" se introducen con el siguiente programa BASIC:

```
 10 REM *** CIRCULOS ***
 20 REM Juan Mtz.Velarde,1984
 30 CLEAR 28650: LET T=0
 40 FOR F=28672 TO 28723
 50 READ A: POKE F, A
 60 LET T=T+A: NEXT F
 70 IF T< >6491 THEN PRINT "ERROR EN DATAS" : STOP
 80 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
    MAQUINA ": PAUSE 0
 90 RANDOMIZE USR 28672
100 DATA 205 , 107 , 13 , 62 , 2 , 205 , 1 , 22 , 217 , 229 , 217
110 DATA 6 , 0 , 197 , 120 , 205 , 40 , 45 , 193 , 197 , 120 , 205 , 40 , 45 , 193 , 197 ,
    120 , 205 , 40 , 45 , 205 , 45 , 35
120 DATA 193 , 62 , 251 , 219 , 254 , 230 , 8 , 40 , 6 , 4 , 62 , 87 , 184 , 32 , 221
130 DATA 217 , 225 , 217 , 201
```

13

RELABEL

La tercera parte del libro presenta una serie de programas y rutinas en código máquina, muy ricos en enseñanza, que pueden ser usados en tus propios programas, modificándolos según tus necesidades.

Este capítulo está dedicado a un solo programa, escrito 100 en código máquina. Su nombre es "RELABEL" y pertenece al conjunto de programas de "Utilidad". Su función es renumerar una a una todas las sentencias de un programa BASIC, sea cual sea su longitud. El usuario puede definir el número de la sentencia de comienzo y la diferencia entre sentencia y sentencia.

La verdadera ventaja de este programa es que detiene automáticamente la rutina si se sobrepasa el número de línea 9999, retornando al BASIC. La rutina no afecta a la numeración de órdenes GOTO y GOSUB, que deben ser modificados "a mano" por el usuario.

Para comprender por qué y cómo funciona el programa "RELABEL" debemos analizar su estructura assembler. Este es el listado:

===

Especificaciones : "RELABEL" para ZX Spectrum 16K/48K.
Descripciòn General : El programa renumera cada sentencia de un programa BASIC.
Longitud : 142 bytes.
Entrada : Ningùn requerimiento.
Salida : Ningùn requerimiento.
Registros Usados : A , B , C , D , E , H y L.

```
5B00                      ORG 7000H          ;
5B00   2A535C                 LD HL, (PROG)      ; HL = INICIO BASIC
5B03   ED754B5C               LD BC, (VARS)      ; BC = FIN BASIC + 1
5B07   110A00                 LD DE,10D          ; DE = LINEA INICIAL
5B0A   C5              SIG    PUSH BC            ; ALMACENA BC
5B0B   72                     LD (HL),D          ; MODIFICA BYTE ALTO
5B0C   23                     INC HL             ;
```

```
5B0D    73        00170          LD (HL),E             ; MODIFICA BYTE BAJO
5B0E    23        00180          INC HL                ;
5B0F    4E        00190          LD C, (HL)            ; C = BYTE BAJO LONG.
5B10    23        00200          INC HL                ; B = BYTE ALTO LONG.
5B11    46        00210          LD B, (HL)            ;
5B12    23        00220          INC HL                ;
5B13    09        00230          ADD HL,BC             ; HL = SENTENCIA SIG.
5B14    010A00    00240          LD BC,10D             ; BC = PASO
5B17    79        00250          LD A,C                ;
5B18    83        00260          ADD A,E               ;
5B19    5F        00270          LD E,A                ; E = E + C
5B1A    78        00280          LD A,B                ;
5B1B    8A        00290          ADC A,D               ;
5B1C    57        00300          LD D,A                ; D = D + B
5B1D    C1        00310          POP BC                ; RECUPERA VARS
5B1E    22B05C    00320          LD (23728), HL        ; GUARDA EL PUNTERO
5B21    21F0D8    00330          LD HL,55536           ; HL = 65536- 10000
5B24    19        00340          ADD HL,DE             ; HL = HL + DE
5B25    380F      00350          JR C,ERR              ; SI DE > 9999 --> ERR
5B27    2AB05C    00360          LD HL, (23728)        ; RECUPERA PUNTERO
5B2A    22B05C    00370          LD (23728),HL         ; Y GUARDALO OTRA VEZ
5B2D    AF        00380          XOR A                 ; INDIC ARRASTRE = 0
5B2E    ED42      00390          SBC HL,BC             ; ¿ HEMOS ACABADO ?
5B30    C8        00400          RET Z                 ; RETORNA SI ES ASI
5B31    2AB05C    00410          LD HL, (23728)        ; RECUPERA PUNTERO
5B34    18D4      00420          JR SIG                ; PROCESA LA SIGUIEN-
                                                       ; TE LINEA
5B36    3E02      00430  ERR     LD A,2                ;        ABRE
5B38    CD0116    00440          CALL OPEN             ; CANAL NUMERO DOS
5B3B    11445B    00450          LD DE,23364 D         ; DIRECCION MENSAJE
5B3E    014A00    00460          LD BC,74 D            ; LONGITUD MENSAJE
5B41    C33C20    00470          JUMP PR-STRING        ; IMPRIMELO
5B44              00480          DEFB 16,2,13,18,1     ; AT 2,13; FLASH 1
5B49              00490          DEFM "ERROR"'         ;.
5B4E              00500          DEFB 18,0,13,13       ; FLASH 0  ' '
5B52              00510          DEFM "El n. de        ;
                                   linea excede la     ;
                                   9999.               ;
5B70              00520          DEFB 13               ;
5B71              00530          DEFM "Modifica        ;
                                   linea inicial o     ;
                                   paso"               ;
0000              00540          END                   ;

1601    OPEN
203C    PR-STRING
```

5B0A SIG
5B36 ERR

==

El programa "RELABEL" hace uso de todos los registros del microprocesador. Cada uno cumple una o varias funciones:

Registro A: Aquí se utiliza en un modo general, sin una función determinada.

Registro BC: Almacena la dirección y memoria de FIN del área BASIC o el valor entre sentencia y sentencia, que a partir de ahora llamaremos paso.

Registro DE: Almacena durante todo el programa el valor que debe ser introducido como número de línea de la próxima sentencia BASIC.

Registro HL: Se utiliza como puntero. Almacena la dirección de memoria donde se guardan los datos a modificar (la numeración de cada línea) .

Ciertos contenidos se almacenan temporalmente en el Stack o en la dirección de memoria 5C80H / 23728 d.

El registro HL se inicializa el primero con la dirección de memoria que determina el comienzo del área BASIC en memoria. En esta zona se encuentra la numeración de cada línea, dato que hay que cambiar. El registro BC se inicializa con el comienzo del área de variables, o lo que es lo mismo, el fin del área BASIC. Una y otra zona son limítrofes. Dos variables del sistema, PROG y VARS, almacenan en las direcciones de memoria 5C53h / 5C54 h y 5C4B h / 5C4Ch, respectivamente los valores de comienzo de una y otra zona.

El registro DE se inicializa con el valor 10d. Este será el primero número de línea del programa BASIC. DE se incrementa a medida que se procesan otras sentencias.

Es fundamental conocer la estructuración de información del área BASIC. A continuación daremos un pequeño repaso a este área. Una explicación más detallada se encuentra en el apartado séptimo del tercer capítulo.

Para almacenar una línea BASIC hace falta un mínimo de 6 bytes. Los dos primeros bytes guardan el número de la línea. Este es el único caso en que el ordenador almacena un número entero en la forma byte alto-byte bajo, siendo la inversa la forma normal.

Los dos bytes siguientes, ya en forma byte bajo-byte alto, almacenan la longitud de esa línea, medida en bytes, desde el quinto byte al final.

El quinto byte es el correspondiente al código del comando de la línea. Cada comando tiene su código correspondiente. Entre éste y el último byte se encuentra todo lo que realmente forma una sentencia, y que está visible en un listado. El último byte determina el fin de la sentencia. Siempre es el código 0Dh /13d.

Precisamente las líneas assembler 00150 y 00170 introducen el valor de DE en los dos primeros bytes de la primera línea.-Se hace uso del puntero HL y el direccionamiento por registro para modificar la numeración. El número de línea de la primera sentencia será el que se haya introducido en DE anteriormente. Si se inicializa con 100d, éste será el primer valor. Observa cómo, al ser una excepción, se introduce primero el byte alto del registro (D) y en la siguiente dirección el byte bajo (E).

El registro HL es incrementado entretanto para pasar de una a otra dirección.

Esta es la parte que cambia la numeración de cada línea. La siguiente vez que se procesen estas instrucciones, el registro DE contendrá otro valor, que será el número de la siguiente línea.

Si te has fijado, anteriormente se ha almacenado el registro BC en el Stack. La razón es que este mismo registro será utilizado más adelante para otro propósito y su contenido no debe perderse. El programa introduce en BC el contenido de los bytes tercero y cuarto. Estos dos bytes almacenan la longitud de la línea. Observa como esta vez es el byte menos significativo (C) el primero que se utiliza. El byte más significativo aparece en segundo lugar y se introduce en el registro B. Tras cada una de estas instrucciones, el puntero HL se incrementa en uno.

La suma del puntero (HL) y de la longitud (BC) dará como resultado la dirección de memoria que almacena el número de línea de la siguiente instrucción. Este resultado se almacenará en el registro HL.

Repasemos el contenido actual de los registros: Los registros A, B y C contiene información nada importante. El primer registro no ha sido usado todavía y el BC contiene la longitud de la sentencia anterior. El registro DE contiene el número de línea de la sentencia anterior. El registro restante, el HL, contiene la dirección de memoria que almacena el número de línea de la siguiente sentencia.

A continuación debe sumarse al registro DE el paso que deseemos. El resultado será el número de línea de la siguiente sentencia. Si queremos que las sentencias vayan de 10 en 10, el paso debe ser igual a 10. Cualquier otro paso puede utilizarse a elección del usuario (10, 100, 435, 1000, 5023...).

Precisamente esta operación es la que se realiza a continuación. El registro BC almacena el paso deseado (en RELABEL es igual a 10), que debe ser sumado a DE. El único registro simple que puede sumar dos valores es el Acumulador. Se realiza del siguiente modo:

El registro A se carga con el valor que contenga el registro C ya aquel se le suma el valor del registro E. Los registros C y E almacenan los bytes bajos de BC y DE, respectivamente. El resultado de la suma se traspasa al registro E.

La misma operación se realiza con los dos bytes altos, B y D. La única diferencia es que esta vez se hace uso de la instrucción ADC (Suma con el indicador de Arrastre) , por si existió un arrastre en la suma de A y E. De esta manera, a DE se le ha sumado el registro BC, y ahora contiene el que será próximo número de línea.

Seguidamente es necesario saber si la suma anterior ha sobrepasado el valor 9999, que corresponde al mayor número de línea BASIC en un programa del Spectrum, pues si este es el caso, aparecerían errores en el programa y no podría funcionar .

He utilizado para ello un sistema utilizado en la misma ROM. Este método hace uso del registro HL y no llega a modificar el registro DE. Recuerda que HL contiene un valor importante: la dirección de memoria de la que sería próxima sentencia. Por ello se almacena antes de llegar a ejecutarlo en la dirección de memoria 5CB0 h y 5CB1h /23728 d y 23729 d. Son dos direcciones de memoria no usadas entre las variables del sistema. Su contenido no puede ser borrado por un programa BASIC o código máquina.

El sistema se basa en cargar el registro HL con el que sería el valor máximo (9999), restado de FFFFh / 65535 d (65535 - 9999 = 55536). A este valor se le suma el contenido de DE. Si el resultado es mayor a 65535, es señal que el valor de DE era mayor a 9999. En este caso, el indicador de arrastre toma el valor 1. Un salto relativo condicional de indicador de arrastre detiene el programa, desviándolo a otra parte denominada ERR (de ERROR). Nótese que en esta operación no se modifica el valor de DE, sino el de HL. El registro DE sigue manteniendo el número de la próxima sentencia.

En caso de no haber superado DE el valor 9999d, se procede a averiguar si se ha finalizado con todas las sentencias BASIC. En ese caso, sería necesario retornar al BASIC. El programa habría terminado su función.

El registro HL es restaurado a su valor original, almacenado en la dirección 5CB0h y 5CB1 h. Seguidamente, se vuelve a almacenar en la misma dirección de memoria. En caso de tener que continuar con el programa, será este valor el que debe contener HL.

Si se ha finalizado con todas las sentencias BASIC, el registro HL debe ser igual al comienzo del área de variables, zona limítrofe con la de BASIC. Al principio del programa se inicializaba BC con el comienzo del área BASIC. Si se ha acabado con todas las sentencias BASIC, HL y BC deben ser iguales.

El registro BC se guardó en el Stack con aquel valor. Si te fijas bien en el listado, verás que un POP BC se encuentra pocas instrucciones atrás. Ahora tienen HL y BC los valores adecuados. El segundo debe ser restado al primero, pues la instrucción de resta con registros dobles sólo puede ser realizado con el HL.

La instrucción existe únicamente bajo la forma "SBC HL,BC" que utiliza el indicador de arrastre. Para asegurarnos que éste es igual a cero y no hay ningún fallo en la sustracción, se utiliza la instrucción XOR A. Su verdadera función aquí no es poner a cero el Acumulador, sino poner a cero el indicador de arrastre.

Si HL es igual a BC, el resultado será igual a cero y el indicador de cero tomará el valor uno. Un retorno incondicional finalizará el programa.

En caso contrario, HL debe ser restaurado a su valor anterior, que se encuentra en la dirección 5CB0h y 5CB1 h. El registro DE contiene el número de línea de la próxima instrucción. El resto de los registros contienen valores sin importancia.

El programa vuelve a la parte etiquetada con SIG, para procesar la siguiente sentencia. Los registros HL y DE contienen los valores adecuados para hacerlo.

Todo lo explicado anteriormente se repetirá hasta que se haya acabado con todas las sentencias o hasta que se sobrepase el valor 9999d. En este último caso, el programa salta a la parte etiquetada con "ERR", que tiene la única función de imprimir un mensaje de error en la pantalla.
En este caso debe modificarse la línea inicial (inicialización de DE) o el paso (segunda inicialización de BC, línea assembler 00240). Seguramente es esto último la causa del error. Son las siguientes direcciones de memoria las que almacenan los valores de línea inicial y de paso:

LINEA INICIAL Byte Bajo = 23304d / 5B08h
Byte Alto = 23305d / 5B09h

PASO Byte Bajo = 23317d / 5B15h
Byte Alto = 23318d / 5B16h

Si se quiere modificar la línea inicial a 5, debe teclearse POKE 23304,5: POKE 23305,0, pues 5 + 0 * 256 = 5.

Si se quiere modificar el paso a 1000, debe teclearse POKE 23317,232 : POKE 23318, 3, pues 232 + 3 * 256 = 1000.

* * *

El programa se almacena en la memoria intermedia de la impresora para no llegar a interferir de ningún modo posible un programa BASIC. El buffer de impresora ocupa direcciones de memoria inferiores a las del área BASIC. De esta manera, la longitud del programa no es un impedimento para el funcionamiento de "RELABEL".

Puedes probar la rapidez de " RELABEL " con el mismo programa cargador, probando con las combinaciones de línea inicial y paso que desees. Si quieres renumerar otros programas, carga en memoria primero el programa "RELABEL", da RUN y carga entonces el que desees, sencillamente con LOAD " ". El programa se pone en marcha con el comando RANDOMIZE USR 23296.

```
10 REM *** RELABEL ***
20 REM Juan Mtz.Velarde,1984
30 LET T=0
```

```
 40 FOR F=23296 TO 23437
 50 READ A: POKE F,A
 60 LET T=T+A: NEXT F
 70 IF T<>11470 THEN PRINT "ERROR EN DATAS": STOP
 80 DATA 42 , 83 , 92 , 237 , 75 , 75 , 92 , 17 , l0 , 0
 90 DATA 197 , 114 , 35 , 115 , 35 , 78 , 35 , 70 , 35 , 9
100 DATA 1 , l0 , 0 , 121 , 131 , 95 , 120 , 138 , 87
110 DATA 193 , 34 , 176 , 92 , 33 , 240 , 216 , 25 , 56 , 15 , 42 , 176 , 92
120 DATA 34 , 176 , 92 , 175 , 237 , 66 , 200 , 42 , 176 , 92
130 DATA 24 , 212 , 62 , 2 , 205 , 1 , 22
140 DATA 17 , 68 , 91 , 1 , 74 , 0 , 195 , 60 , 32
150 DATA 22 , 2 , 13 , 18 , 1 , 69 , 82 , 82 , 79 , 82 , 18 , 0 , 13 , 13
160 DATA 69 , 108 , 32 , l10 , 46 , 32 , l00 , l01 , 32 , 108 , 105 , l10 , 101 , 97 ,
    32 , l0l , 120 , 99 , l0l , l00 , 101 , 32 , l08 , 97 , 32 , 57 , 57 , 57 , 57 , 46
170 DATA 13 , 77 , 111 , l00 , l05 , l02 , l05 , 99 , 97 , 32 , 108 , l05 , 110 , l01 ,
    97 , 32 , l05 , 110 , l05 , 99 , l05 , 97 , l08 , 32 , 111 , 32 , 112 , 97 , 115 , 111
```

14

El programa REPLICANTE

El nombre del programa indica ya su función: replicar, en el sentido de hacer réplicas, de repetir algo que se ha dicho o hecho. El programa REPLICANTE es un ejemplo del uso de las rutinas LOAD o SAVE del sistema operativo, de la ROM. Permite cargar un programa (con la rutina de LOAD) y grabarlo en una cinta magnética (con la rutina de SAVE). El programa salvado será una réplica del cargado anteriormente. El programa que se desee replicar se almacena entretanto en la memoria del ordenador .

El usuario puede disponer de una réplica de seguridad, de manera que si alguna de sus cintas se extravía, recuperará su contenido, si fue replicado anteriormente.

No importa si el programa está protegido y no se puede acceder al listado, o si empieza automáticamente. Es totalmente indiferente si se trata de un programa BASIC, de una matriz numérica, de una matriz alfanumérica o de un programa en código máquina. ¡Todo ello puede ser replicado!

Los que gusten de investigar y analizar software sabrán que han aparecido en el mercado ciertos programas con una característica especial: carecen de la parte inicial de la grabación que contiene los datos del programa que viene a continuación. El programa REPLICANTE aporta la gran ventaja de tratar también estas excepciones y realizar réplicas exactas a ellas.

A continuación aparece el listado assembler, que será analizado en su totalidad:

```
=======================================================================

Especificaciones : "REPLICANTE" para ZX Spectrum 16K/48K.
Descripciòn General : Utilizando las rutinas LOAD y SAVE de la
memoria ROM se realizan rèplicas exactas de cualquier programa.
Longitud : 232 bytes.
Entrada : Ningùn requerimiento.
Salida : Ningùn requerimiento.
Registros Usados : A, B, C, D, E, H, L, IX en programa REPLICANTE
                   A, B, C, D, E, H, L, IX en rutinas ROM
```

```
5B00            00100          ORG 5B00H          ;
5B00  CD6B0D    00110          CALL CLS           ; BORRA PANTALLA
5B03  3E02      00120          LD A,2 D           ; ABRE
5B05  CD0116    00130          CALL OPEN          ; CANAL NUMERO DOS
5B08  11755B    00140          LD DE,MENS         ; DIRECCION MENSAJE
5B0B  017300    00150          LD BC,115 D        ; LONGITUD MENSAJE
5B0E  CD3C20    00160          CALL PR-STRING     ; IMPRIMELO
5B11  3EFE      00170  MENU    LD A,254 D         ; SEMIFILA CAPS - V
5B13  DBFE      00180          IN A, (FEH)        ; LEE LA SEMIFILA
5B15  CB5F      00190          BIT 3,A            ; ¿TECLA "C" PULSADA?
5B17  2819      00200          JR Z,LOAD          ; SI -> LOAD
5B19  CB57      00210          BIT 2,A            ; ¿TECLA "X" PULSADA?
5B1B  2822      00220          JR Z,LOADDATA      ; SI -> LOADDATA
5B1D  3EFD      00230          LD A,253D          ; SEMIFILA A - G
5B1F  DBFE      00240          IN A, (FEH)        ; LEE LA SEMIFILA
5B21  CB4F      00250          BIT 1,A            ; ¿TECLA "S" PULSADA?
5B23  2835      00260          JR Z,SAVE          ; SI -> SAVE
5B25  CB47      00270          BIT 0,A            ; ¿TECLA "A" PULSADA?
5B27  283D      00280          JR Z,SAVEDATA      ; SI -> SAVEDATA
5B29  3EFB      00290          LD A,251D          ; SEMIFILA Q - T
5B2B  DBFE      00300          IN A, (FEH)        ; LEE LA SEMIFILA
5B2D  CB5F      00310          BIT 3,A            ; ¿TECLA "R" PULSADA?
5B2F  20E0      00320          JR NZ,MENU         ; NO -> MENU
5B31  C9        00310          RET                ; RETORNO SI PULSADA
5B32  37        00320  LOAD    SCF                ; Set Carry Flag
5B33  3E00      00330          LD A,0 D           ; SEÑAL DE CABECERA
5B35  DD21ED5B  00340          LD IX,CABECERA     ; IX = "BASE ADDRESS"
5B39  111100    00350          LD DE,17 D         ; CARGA 17 BYTES
5B3C  CD5605    00360          CALL LD-BYTES      ; REALIZA OPERACION
5B3F  37        00370  LOAD    SCF                ; Set Carry Flag
                       DATA                       ;
5B40  3EFF      00380          LD A,255 D         ; SEÑAL DATOS
5B42  DD21005E  00390          LD IX,REPLICA      ; CARGA EL MAXIMO DE
5B46  11FFFF    00400          LD DE,65535 D      ; BYTES A PARTIR DE
5B49  CD5605    00410          CALL LD-BYTES      ; LA DIREC. 5E00H
5B4C  DDE5      00420          PUSH IX            ; TRANSFIERE EL VALOR
5B4E  El        00430          POP HL             ; DE IX A HL
5B4F  11005E    00440          LD DE,REPLICA      ; PRIMERA DIRECCION
5B52  AF        00450          XOR A              ; INDIC.ARRASTRE = 0
5B53  ED52      00460          SBC HL,DE          ; REALIZA LA RESTA
5B55  22EB5B    00470          LD (LONG),HL       ; ALMACENA RESULTADO
5B58  18B7      00480          JR MENU            ; VUELVE AL MENU
5B5A  21015E    00490  SAVE    LD HL,REPLICA+1    ; HL CONTIENE 5E01 H
5B5D  DD21ED5B  00500          LD IX,CABECERA     ; DATOS PARA CABECERA
5B61  CD7009    00510          CALL SA-CONTRL     ; EN 5BED - GRABA !
5B64  18F2      00520          JR 5B58 H          ; VUELVE AL MENU
5B66  3EFF      00530  SAVE    LD A,255 D         ; SEÑAL DATOS
                       DATA                       ;
```

```
5B68    DD21015E    00540           LD IX,REPLICA+1         ; IX CONTIENE 5E01 H
5B6C    ED5BEB5B    00550           LD DE, (LONG)           ; DE GUARDA LONGITUD
5B70    CDC204      00560           CALL SA-BYTES           ;  GRABA !
5B73    18EF        00570           JR 5B64 H               ; VUELVE AL MENU
5B75                00580   MENS    DEFB 22,0,10,20,1       ; AT 0,10  ;  INVERSE 1
5B7A                00590           DEFM "REPLICANTE"       ;
5B84                00600           DEFB 22,3,5             ; AT 3,5
5B87                00610           DEFM "C"                ;
5B88                00620           DEFB 20,0               ; INVERSE 0
5B8A                00630           DEFM " LOAD"            ;
5B8F                00640           DEFB 22,5,5,20,1        ; AT 5,5 ; INVERSE 1
5B94                00650           DEFM "X"                ;
5B95                00660           DEFB 20,0               ; INVERSE 0
5B97                00670           DEFM " LOAD Data"       ;
5BAl                00680           DEFB 22,7,5,20,1        ; AT 7,5; INVERSE 1
5BA6                00690           DEFM "S"                ;
5BA7                00700           DEFB 20,0               ; INVERSE 0
5BA9                00710           DEFM " SAVE"            ;
5BAE                00720           DEFB 22,9,5,20,1        ; AT 9,5; INVERSE 1
5BB3                00730           DEFM "A"                ;
5BB4                00740           DEFB 20,0               ; INVERSE 0
5BB6                00750           DEFM " SAVE Data"       ;
5BC0                00760           DEFB 22,11,5,20,1       ; AT 11,5; INVERSE 1
5BC5                00770           DEFM "R"                ;
5BC6                00780           DEFB 20,0               ; INVERSE 0
5BC8                00790           DEFM " RET"             ;
5BCC                00800           DEFB 22,15,2            ; AT 15,2
5BCF                00810           DEFM "Por Juan          ;
00820                               Mtz.Velarde,1984"       ;
0000                00830           END                     ;

04C2    SA-BYTES
0556    LD-BYTES
0970    SA-CONTRL
0D6B    CLS
1601    OPEN
203C    PR-STRING
5B11    MENU
5B32    LOAD
5B3F    LOAD DATA
5B5A    SAVE
5B66    SAVE DATA
5B75    MENS
5BEB    LONG
5BED    CABECERA
5E00    REPLICA
5E01    REPLICA+1
=====================================================================
```

Para comprender el programa "REPLICANTE" debemos conocer su estructura. Consta de dos partes muy bien diferenciadas. La primera de ellas se ocupa de imprimir el MENU en pantalla y posibilitar el acceso a cuaiquiera de las cinco opciones que REPLICANTE ofrece. La segunda parte la forman las instrucciones pertenecientes a las cinco opciones. Cada una de estas partes guarda una estructura interna, que será analizada totalmente y que es la siguiente:

Parte primera:

a) Borrado de pantalla y apertura del CANAL 2.
Impresión del MENU.

b) Acceso a cada una de las 5 opciones del MENU.

Parte segunda:

a) Carga Normal
b) Carga sin cabecera.
c) Grabación Normal.
d) Grabación sin cabecera.
e) Retorno al BASIC.

El borrado de pantalla y la apertura del canal dos se realiza con el mismo sistema que hemos venido utilizando, las rutinas CLS y OPEN. Lo realmente importante viene ahora: la impresión del menú y sus opciones. Para ello se hace uso de la rutina ROM PR-STRING. El registro doble DE almacena la dirección de memoria a partir de la cual se encuentran los códigos que corres-ponden a los caracteres del menú. Esta dirección es la 5B75 h/ 23413 d. Los códigos forman un "banco o agrupación de datos" y se encuentran tras el programa, que finaliza en la dirección 5B74 h /23412 d.

El número de caracteres que deben ser impresos se expresa con el contenido del registro BC y que es igual a 72 h /114 d. La rutina PR-STRING, ubicada en la dirección de memoria 203C h, en plena ROM, no necesita más datos e imprime el texto en la pantalla. Este consta del nombre del programa y de cada opción, dispuestos en la pantalla y en diferentes atributos gracias al uso de los caracteres de control.

El acceso a cada una de las cinco opciones antes mencionadas se realiza mediante la pulsación de una tecla determinada. Este es un uso práctico del contenido del primer apartado del Capítulo 12, referido al Teclado.

La tecla "C" accede a la rutina de carga normal. La tecla "X" accede a la rutina de carga de programas sin cabecera. La diferencia entre unos y otros y la significación de la cabecera serán explicados más adelante. Otras dos teclas, la "S" y la " A" permiten realizar un SAVE con o sin cabecera, respectivamente. Se ha reservado una tecla para el retorno al BASIC, la "R".

Observa que las teclas para carga y grabación se han escogido de tal manera que pertenezcan a la misma semifila, en grupos de dos. Esta disposición de las teclas resta belleza al conjunto, pero

aporta una gran ventaja, el teclado debe ser leído una sola vez por la instrucción IN para comprobar el estado de dos teclas. En esta operación es importante recordar que se utiliza una instrucción BIT y no AND. Ambas ocupan en si la misma memoria, pero la primera no modifica el contenido del Acumulador.

En el listado assembler podrás ver que al principio se lee la semifila CAPS-V, donde se encuentran las teclas C y X. El resultado de esta lectura se almacena en el registro A. La instrucción BIT 3, A comprueba si la tecla "C" ha sido pulsada. Como ya vimos en su momento, si este es el caso, el bit número 3 tendrá el valor 0, el indicador de cero tomará el valor 1 y se procesará un salto relativo hacia la rutina de carga (etiquetada con LOAD).

Recuerda que esta instrucción no modifica en ningún momento el valor del Acumulador. De este modo, si no se cumple la condición anterior, puede pasarse a comprobar el estado de la tecla "X", cuyo contenido lo almacena el bit número 2. Con esta ocurre lo análogo. Si se pulsa, se ejecuta un salto a la parte etiquetada con LOADDATA.

Si se hubiera escogido la instrucción AND, hubiéramos necesitado de cuatro bytes más, necesarios para introducir en el Acumulador el valor de entrada de la semifila y leer su estado. AND modifica el contenido del registro A al ser ejecutada.

Las teclas que acceden a las rutinas de grabación ("S" y " A ") pertenecen a otra semifila y hay que volver a realizar la instrucción de IN. En este caso son los bits número 1 y 0 los que almacenan el estado de las teclas "S" y "A", respectivamente. Las rutinas están etiquetadas con "SAVE" y "SAVEDATA", para salvar programas con o sin cabecera.

En la última opción (retorno al BASIC) es totalmente indiferente el uso de BIT o AND. Aunque se lea toda la semifila, sólo nos interesa el estado de una tecla determinada, de un bit determinado. Por ello es igual utilizar BIT 3, A para conocer si la tecla "R" ha sido pulsada, o AND 08h, que cumple la misma función. Ambas instrucciones ocupan dos bytes en memoria.

Observa que esta última opción, el salto relativo es condicional de indicador de cero diferente de cero. Esto significa que el salto se ejecutará sólo en el caso que el indicador contenga el valor 1. Si la tecla "R" no ha sido pulsada, el salto se ejecuta al comienzo del Menú, a la parte etiquetada con este nombre.

De esta manera consigue un bucle que "espera" a que se pulse una tecla de entre 5. Cuatro de ellas harán acceder al usuario a las diferentes rutinas de carga y grabación. Una de ellas le devolverá al BASIC, pues si se pulsa la "R", el salto no se llevará a cabo y si la instrucción siguienteaél, un RET.

Son las cuatro rutinas de LOAD y SAVE las más importantes de "REPLICANTE", y las que analizaremos a continuación. Al almacenar en cinta magnética un programa o unos datos, el ordenador envía a través de un port determinado los códigos que lo forman, en forma de sonidos. Estos afectan a la cinta y quedan allí almacenados hasta que otros ocupen su lugar. El ordenador reconoce sus propios códigos al cargarlos en la memoria. Tal vez te hayas preguntado cómo puede el ordenador saber si está cargando un programa BASIC, o una serie de Bytes o una matriz, pues cada una de ellas se procesa de forma diferente.

Por tu experiencia en la programación BASIC distinguirás las grabaciones efectuadas por un ZX Spectrum de las de un ZX81, por ejemplo. Es característico del ZX Spectrum dividir la grabación en dos partes. La primera es de corta duración y precede a una pequeña causa. Tras ella viene la segunda parte, cuya longitud depende exclusivamente de la del programa o los datos. Si es un programa largo, esta segunda parte será larga. Al cargarse un programa aparece también esta estructura. Tras la primera parte se imprimen los datos correspondientes al programa. El usuario puede de esta manera saber si le interesa o no la carga.

Es la primera parte la que contiene los datos pertenecientes a la segunda parte. Aquí se especificará el tipo de carga que se está realizando: un programa, bytes, matrices numéricas o matrices alfanuméricas. Parte de esta información la usa el ordenador para informar al usuario (tipo de carga y nombre del programa) y parte se la reserva (la longitud de la segunda parte, comienzo automático o no automático si es un programa BASIC, dimensionamiento de las matrices si la grabación fue de este tipo, etc). A partir de ahora, la primera parte la denominaremos "Cabecera", por estar situada a la cabeza de cualquier grabación. La segunda parte se denominará "Datos" porque son estos y sólo estos los datos de la grabación que forman el programa, los bytes o las variables.

Como ya mencioné anteriormente, han aparecido últimamente en el mercado programas que carecen de cabecera. Los datos necesarios para ella se encuentran en un programa que por fuerza tuvo que ser cargado con anterioridad. Con estas rutinas del LOAD y SAVE sabrás cómo grabar tus propios programas en código máquina sin cabecera, para que queden protegidos frente a usuarios que carezcan de un arma tan útil como puede ser el REPLICANTE.

La primera rutina del programa es la etiquetada LOAD. Su comienzo es en la dirección 5B32h/23346 d. En el listado, la rutina LOAD ocupa las sentencias assembler 00320 a 00360. Tiene la función de cargar la cabecera de un programa en memoria. Los datos para esta cabecera serán usados más adelante por la segunda rutina llamada LOADDATA.

La primera instrucción de esta rutina supone ya un misterio. Recibe el nombre de SCF y, como no, es una abreviatura para "Set Carry Flag", en castellano "Pon a Uno el Indicador de Arrastre": Esta instrucción, cuyo código objeto es de un byte de longitud, afecta a un bit determinado del registro F, el bit que almacena el Indicador de Arrastre. SCF introduce directamente en ese bit el valor 1, no tiene más complicación.

La razón para introducir una instrucción de este tipo, utilizada en casos contados, es que la rutina a la que accede en ROM es compartida por otra función diferente. Poner el Indicador de Arrastre a uno es una señal identificatoria que indica que es el comando LOAD el que debe ser ejecutado.

En la rutina ROM de carga existen dos posibilidades: carga de la cabecera de un programa o de los datos que lo forman. Son dos cosas diferentes que deben ser tratados distintamente. El indicador de carga de cabecera es el valor 0 como contenido del Acumulador. Esta es la función que ejerce la instrucción LD A, 0.

Una de las funciones que ejerce el establecimiento de la pausa en la grabación es "dar tiempo" al ordenador para estudiar la información de la cabecera en la carga del programa. En este tiem-

po, el ordenador puede preparar su memoria de acuerdo al contenido de la cabecera. La primera parte de cualquier grabación "normal" está formada por exactamente 17d bytes. Diez de ellos se reservan al nombre del programa, dos contienen la longitud en bytes de la segunda parte, de los datos, uno almacena el tipo de grabación que es (BASIC, Bytes, Matriz de números o de caracteres) , y los dos restantes guardan información diversa que depende del tipo de grabación efectuada (grabado o no con LINE para comienzo automático, dirección de memoria de comienzo y longitud para CODE, etc.).

El ordenador opera con esta información para cargar los datos satisfactoriamente. Para ello los almacena en un lugar determinado de su memoria, el área de trabajo. Este último caso es el que ocurre al ejecutar un LOAD " " y, cargar un programa. En el caso de REPLICANTE es diferente: no se desea cargar un programa, sino almacenar su cabecera y sus datos en memoria y grabar el conjunto en cinta, haciendo una réplica exacta al original. Por esta razón definiremos nosotros mismos la dirección de memoria donde la información de la cabecera debe cargarse. Estos dstos deben ser utilizados posteriormente por el mismo programa. La dirección de memoria de la que hablamos recibe el nombre de "base address" (dirección base).

A partir de ella se almacenará la información base de un programa. Esta debe ser introducida en el registro IX, de 16 bits. La rutina ROM de carga utiliza este registro para la introducción de información en la memoria. Al cargar la cabecera contiene la dirección de memoria donde su contenido debe ser introducido. El número de bytes a introducir está definido por el registro doble DE y como ya comenté, alcanza el valor de 17d bytes. La dirección de memoria escogida por mi para IX es la 5BED h/ 23533 d. Se encuentra al final de la memoria intermedia de la impresora, sin llegar a alcanzar las variables del sistema. La dirección 23533d y las dieciséis siguientes almacenarán el contenido de la cabecera.

Todos los registros contienen los valores adecuados para proceder a la carga, no de un programa, sino de la cabecera de un programa. Los datos en si se introducirán en la memoria más adelante. El punto de entrada de la rutina de carga es el 0556h/1366 d. Recibe el nombre de LD-BYTES.

Observa que la rutina se accede con un CALL y no con un JUMP. Esta última instrucción retornaría al BASIC al finalizar la rutina de ROM, habiendo cargado los datos de la cabecera de un programa, no el conjunto. Tras terminar la rutina ROM se continúa con otra carga, esta vez la correspondiente a la segunda parte. Si se quiere cargar un programa sin cabecera, es esta parte la que accede desde el principio, pulsando la tecla "X".

Esta es la segunda rutina de carga del programa REPLICANTE. Su nombre es LOADDATA y tiene su comienzo marcado en la dirección de memoria 5B3Fh. Carga los datos correspondientes a un programa con o sin cabecera.

Para ello, ciertos registros contienen valores determinados. La rutina ROM usada es la misma que la anterior, aunque los registros contienen valores diferentes. El indicador de carga es el valor 1 en el Indicador de Arrastre. Por ello la primera instrucción es aquí también SCF. La diferencia fundamental entre una y otra rutina es que en la segunda el Acumulador contiene el valor FFh / 255d. Es el indicador de carga de datos, y no de cabecera, que sería 00,

Con este último dato la rutina LD-BYTES sabe perfectamente que debe cargar en memoria una serie de bytes. Pero la partir de qué dirección y cuántos son los que hay que cargar?

Estos datos se expresan mediante los registros IX y DE. El primero de ellos contiene la dirección de memoria de inicio, a partir de la cual debe cargarse la información de la segunda parte. El registro DE contiene el número de bytes que hay que cargar. El programa que se quiera replicar se almacena en la memoria RAM. Después no hay más que enviar para obtener una réplica. El valor introducido en el registro IX es el 5E00h / 24064d. He escogido esta dirección en consideración a los usuarios de Microdrive e Interface 1, pues direcciones inferiores están ocupadas por datos del Microdrive o del Interface. La Memoria puede albergar programas de la dirección 24064 a la última, la 65535 d para el 48K o la 32767 d para el 16K. La máxima longitud de un programa que REPLICANTE puede replicar a la perfección es de 41471 b para el 48K y de 8704b para el 16K.

Precisamente este dato es el que se debe introducir en el registro DE, la longitud de los datos. Podríamos conocer este dato leyendo los dos bytes correspondientes de la cabecera, almacenados junto al resto de la información de la primera parte a partir de la dirección 5BED. Pero es mucho más sencillo introducir en DE el valor máximo que puede alcanzar, aunque sepamos que no se pueden replicar programas de tal longitud. Aparte de esto, si se ha escogido la opción "X", de carga de programas sin cabecera, no se puede conocer la longitud de los datos. Si resulta que el programa es menor que será el caso más corriente, no ocurrirá nada anormal: simplemente se cargarán tantos datos como el ordenador reciba.

Ya estamos en disposición de llamar a la rutina LD-BYTES. Esta cargará, como es nuestro deseo, el programa en la memoria, a partir de la dirección 24064 d. Al finalizar esta rutina, el registro IX regresa con la dirección de memoria donde fue introducido el último dato. Las instrucciones que vienen a continuación, hasta la dirección de memoria 5B58h, tienen la función de calcular y almacenar la longitud del programa. Si en la carga fue un dato sin importancia, la tiene en la grabación sin cabecera. El ordenador debe saber cuántos bytes tiene que salvar. En la grabación normal, puede aprovechar los datos utilizados de la cabecera, pero no si carece de ella. Por ello la lorlgitud del programa debe ser calculada y almacenada antes de volver al menú.

La cantidad de bytes cargados en memoria puede ser fácilmente calculada si se resta la dirección de memoria de último byte de la dirección de memoria del primer byte. El resultado será siempre un número positivo. El único registro doble con capacidad para realizar restas es el HL. Por ello se traspasa el contenido de IX a HL. De este modo, HL contiene la dirección de memoria del último byte. Primero se almacena el contenido de IX en el stack, valor que es inmediatamente recuperado por HL. El anterior contenido de este registro es borrado. DE se inicializa con la dirección de memoria de comienzo, la 24064 d. Esta dirección se expresa con una etiqueta denominada REPLICA, que indica que a partir de aquella dirección comienzan a almacenarse los datos para la réplica. Casi todo está preparado para la resta. El indicador de arrastre se pone a cero con XOR A y SBC HL, DE realiza la sustracción. El resultado es introducido automáticamente en el registro HL.

La siguiente instrucción almacena el contenido de HL en la dirección de memoria 5EEB h. Esta dirección será posteriormente utilizada para leer el dato. La rutina finaliza ejecutando un salto relativo a la dirección 5B11 h / 23313 d, que corresponde al MENU de opciones.

Ya están analizadas las dos rutinas de carga del programa REPLICANTE. Para grabarlas en cinta disponemos también de dos rutinas. La primera de ellas tiene el comienzo etiquetado con SAVE, en la dirección de memoria 5BA h / 23386 d. Se accede pulsando la opción "S" del MENU Esta vez son otros los registros que deben contener ciertos valores. El registro doble HL se carga con la dirección de memoria a partir de la cual se quieren grabar los bytes MAS UNO. Esta dirección es la resultante de sumar a la etiqueta REPLICA el valor uno y es el contenido de otra etiqueta, llamada REPLICA+ 1. El registro HL se carga, pues con el valor 5E01 h / 24065d.

El registro IX debe contener el "Base Address", la dirección de memoria a partir de la cual se almacenó la información de la cabecera. Es ahora cuando esa información es utilizada. La rutina de grabación se encargará de utilizar los 17 bytes necesarios para formar una cabecera. Estos serán los mismos que entraron al cargarse, de manera que es indiferente el tipo de grabación. La dirección base se inicializó anteriormente con el valor 5BED h / 23533 d. Si se accede a esta opción de grabación normal sin haber cargado previamente un programa, el contenido de la dirección base y los 16 bytes siguientes será igual a 0. Esto confundirá al ordenador y se presentarán problemas al intentar grabar los datos. ¡Pruébalo, el programa no se autodestruirá!

La rutina ROM utilizada se denomina SA-CONTRL y está ubicada a partir de la dirección 0970h. Esta rutina se encarga de grabar tanto la cabecera como los datos, tal y como si fuera un SAVE normal. Esta rutina no presenta el mensaje "Start Tape then ...", sino que comienza automáticamente a grabar. Ten una cinta preparada y en marcha antes de pulsar la "S". Una vez finalizada la réplica, el programa ejecuta un salto relativo negativo a la dirección 5B58 h / 23384 h, que a su vez le hace volver al MENU de opciones. La grabación ha finalizado.

La última rutina de que dispone REPLICANTE es la que se encarga de grabar programas sin cabecera, es decir, sólo los datos. Está ubicada a partir de la dirección 5B66h / 23398 d. Se accede desde el menú pulsando la tecla "A". Recibe el nombre de SAVEDATA.

A diferencia de la anterior son otros los registros que se utilizan. El Acumulador contiene el valor FF h / 255d. Tiene la función de indicar a la rutirla ROM que lo que se quiere grabar son datos, y no una cabecera. Recuerda que en la rutina de carga también se hacía uso de diferentes indicadores para cargar una cabecera o los datos.

Al igual que en la rutina anterior para el registro HL, en esta el registro IX debe contener la dirección de memoria a partir de la cual se encuentran los datos MAS UNO. Estamos hablando de la dirección 5E01 h / 24065 d. Este valor está definido por la etiqueta REPLICA+ 1. El registro DE contiene, como en las rutinas de carga, la longitud del bloque de datos a grabar. Es aquí donde se utilizará el valor calculado y almacenado tras la carga de un bloque de datos, en la rutina LOADDATA. La longitud se calculó restando la dirección del primer elemento de la del último. El resultado se almacenaba en la dirección 5BEB h / 23531 d y 5BEC h / 23532 d, al ser un número de dos bytes. Ahora no hay más que leerlo. El registro DE se carga con el contenido de aquellas direcciones de memoria. El punto de entrada de la subrutina de grabación es el 0402 h / 1218 d, y recibe el nombre de SA-BYTES.

La rutina finaliza al ejecutarse un salto relativo que envía al programa a la dirección 5B64 H / 23396 d, que a su vez ejecuta un segundo salto que accede a la dirección 5B58 h / 23384 d. Des-

de aquí se ejecuta un tercer y último salto al principio del MENU. Este sistema de saltos concatenados ofrece no tener que calcular los desplazamientos completos, sino hasta una determinada dirección de memoria en la que se encuentra una segunda instrucción que de salto que tiene el mismo destino que la anterior .

Los programas que se quieran replicar deben ser cargados en la memoria por partes. Si el programa consta de 3, se cargará la primera y se grabará a continuación, después irá la segunda que se grabará seguidamente. En último lugar se cargará la tercera, con la que se finalizará la réplica.

No se puede cargar un programa sin cabecera e intentar grabarlo con ella. Se equivocarían los datos. Si está permitido grabar un programa con cabecera y "quitársela" con la rutina SAVEDATA. Lo que ocurriría es que necesitaría un cargador previo (del tipo de LOADDATA) para poder ser utilizada sin el REPLICANTE.

Si tú mismo deseas grabar y cargar programas de código máquina en código máquina, básate en las rutinas LOAD (con LOADDATA) y SAVE.

El programa REPLICANTE ocupa relativamente poca memoria. Ejercita el código máquina y haz de REPLICANTE dos versiones: una que sea lo más corta posible y aunque se pierdan funciones del tipo del Menú o las opciones, deben quedar el máximo de bytes libres. La etiqueta REPLICA puede contener un valor más bajo y dejar así más RAM libre. De esta manera habrá pocos programas que no se puedan replicar por problemas de espacio.

La segunda versión no debe reparar en la longitud. El programa debe estar presentado lo mejor que sepas. Mensajes del tipo "Carga satisfactoria" o "Pulsa la Tecla "G" para grabar", etcétera, deben aparecer antes y después del acceso a las rutinas ROM .

Aquí puedes usar el sonido para llamar la atención del usuario al terminar de grabar o de cargar un programa, etc. Por supuesto en esta versión tienes mi permiso para poner tu nombre bajo el mío al imprimirse el menú.

¡Buenas réplicas!

```
10 REM *** REPLICANTE ***
20 REM Juan Mtz.Velarde,1984
25 CLEAR 65535: LET T=0 : REM CLEAR 32767 PARA 16K
30 FOR F=23296 TO 23527
40 READ A: POKE F, A
45 LET T=T+A: NEXT F
50 IF T< >19621 THEN PRINT "ERROR EN DATAS" : STOP
60 DATA 205 , 107 , 13 , 62 , 2 , 205 , 1 , 22 , 17 , 117 , 91 , 1 , 115 , 0 , 205 , 60 , 32
70 DATA 62 , 254 , 219 , 254 , 203 , 95 , 40 , 25 , 203 , 87 , 40 , 34 , 62 , 253 , 219 ,
   254 , 203 , 79 , 40 , 53 , 203 , 71 , 40 , 61 , 62 , 251 , 219 , 254 , 203 , 95 , 32 ,
   224 , 201
80 DATA 55 , 62 , 0 , 221 , 33 , 237 , 91 , 17 , 17 , 0 , 205 , 86 , 5
```

```
 90 DATA 55 , 62 , 255 , 221 , 33 , 0 , 94 , 17 , 255 , 255 , 205 , 86 , 5 , 221 , 229 ,
    225 , 17 , 0 , 94 , 175 , 237 , 82 , 34 , 235 , 91 , 24 , 183
100 DATA 33 , 1 , 94 , 221 , 33 , 237 , 91 , 205 , 112 , 9 , 24 , 242
110 DATA 62 , 255 , 221 , 33 , 1 , 94 , 237 , 91 , 235 , 91 , 205 , 194 , 4 , 24 , 239
120 DATA 22 , 0 , 10 , 20 , 1 , 82 , 69 , 80 , 76 , 73 , 67 , 65 , 78 , 84 , 69
130 DATA 22 , 3 , 5 , 67 , 20 , 0 , 32 , 76 , 79 , 65 , 68
140 DATA 22 , 5 , 5 , 20 , 1 , 88 , 20 , 0 , 32 , 76 , 79 , 65 , 68 , 32 , 68 , 97 , 116 , 97
150 DATA 22 , 7 , 5 , 20 , 1 , 83 , 20 , 0 , 32 , 83 , 65 , 86 , 69
160 DATA 22 , 9 , 5 , 20 , 1 , 65 , 20 , 0 , 32 , 83 , 65 , 86 , 69 , 32 , 68 , 97 , 116 , 97
170 DATA 22 , 11 , 5 , 20 , 1 , 82 , 20 , 0 , 32 , 82 , 69 , 84
180 DATA 22 , 15 , 2 , 80 , 111 , 114 , 32 , 74 , 117 , 97 , 110 , 32 , 77 , 116 , 122 ,
    46 , 86 , 101 , 108 , 97 , 114 , 100 , 101 , 44 , 49 , 57 , 56 , 52
```

15

Listado de Variables

Este será el último capítulo del libro y constituye la culminación de toda la obra. Hemos empezado por lo básico, estudiando los sistemas de numeración del microprocesador y su forma de almacenar valores, las instrucciones más sencillas, conociendo la estructura de la memoria. A continuación vinieron una serie de instrucciones mnemónicas, a las que se dedicaba un capítulo a cada una. La tercera parte se dedica casi exclusivamente al código máquina práctico, ejemplos que muestran el uso del lenguaje assembler.

Este último capítulo contiene el programa assembler más largo del libro, más de 500 bytes de información. Aparte de servirte como objeto de estudio o análisis, lo puedes utilizar en tus propios programas, convirtiéndose así en una utilidad de programación.

Tal y como se indica el título del capítulo, el programa tiene la función de listar todas aquellas variables que estén almacenadas en la memoria del ordenador. Se procesan todo tipo de variables, a saber, variables numéricas de una sola letra, variables numéricas de más de una letra (el ordenador marca una diferencia entre éstas y aquéllas) , variables alfanuméricas, variables de control de bucles, matrices numéricas, matrices alfanuméricas.

Cada una de estas variables se almacena de forma diferente en la memoria. Su naturaleza se indica con el estado de ciertos bits.

Para poder listar todas las variables, debemos conocer el tipo al que corresponden. Debemos conocer por tanto los bits indicadores.

Imaginemos cinco posibles variables, que corresponden con los cinco posibles tipos nombrados anteriormente y observemos su estructura de almacenamiento:

(1) VARIABLE NUMERICA DE UNA SOLA LETRA: A ,VALOR 0 d

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
97d (0 1 1 0 0 0 0 1 b) Carácter. de "a"	0	0	0	0	0

Para almacenar una variable numérica de sólo una letra se necesitan 6 direcciones de memoria. Cada una de estas direcciones contendrá un byte con una función determinada. El primero de ellos almacena el nombre de la variable y la identifica como numérica de letra única. Los tres últimos bits de este byte (bits 7, 6 y 5) contienen siempre los valores 0 1 1. Si el ordenador analiza el área de variables y encuentra un byte con los tres últimos bits con valor 0 1 1, sabe que se trata de una variable numérica. El nombre de la variable lo forman el resto de los bits.

Si no reemplazamos los bits 5, 6 y 7, los llamados identificadores o discriminadores, el byte tendrá la forma 0 1 1 0 0 0 0 1 b, cuyo equivalente decimal es el 97 d. ¿Será casualidad que precisamente el código 97 d corresponde al carácter de "a"? Realmente no es casualidad, pues éste es el método que el ordenador utiliza para almacenar la denominación de las variables. Nótese que el código del carácter " A " es el 65 d, cuya forma binaria es 0 1 0 0 0 0 0 1 b y que los hits 0 a 4 coinciden con los del código 97 d. Esta es la razón por la que el ZX Spectrum no diferencia entre variables escritas en mayúscula o minúscula.

Cuando se requiere del ordenador la impresión de una variable numérica determinada, busca en el área de variables, fijándose siempre en los tres últimos bits de los bytes principales. Si alguno concuerda con el valor 0 1 1, entonces compara el resto de los bits con los del código del carácter de la variable a buscar (bits 00001). Si estos coinciden, la habrá encontrado. Si no, tendrá que seguir buscando.

Cuando antes mencioné los "bytes principales", me refería a los que almacenan los bits indicadores de variable. El ordenador no se fija en los que guardan el contenido de las variables, sólo en los que las definen.

El resto de los bytes, cinco en total, tienen la función de almacenar el contenido de la variable. Para ello se utiliza un sistema muy complejo, y que pertenece a programación assembler avanzada. El número que se debe almacenar se divide en un exponente y una mantisa, ¡pero no en decimal, sino en binario! El segundo byte contiene el exponente y los otros cuatro la mantisa. Quizá hayas oído hablar de los números en coma flotante, que es el nombre que recibe este método.

El ordenador hace uso de un método alternativo para almacenar números en el rango 0 a + 65535, los números llamados "enteros". Se trata de otra forma de almacenar números con los 5 bytes de dato. Más adelante veremos cómo averiguar el valor decimal equivalente que pueda ser comprendido por nosotros.

(2) VARIABLE NUMERICA DE MAS DE UNA LETRA: ABC , VALOR 0

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
161 d	98 d	227 d					
10100001	01100010	11100011 b	0	0	0	0	0
Carácter	Carácter	Carácter					
"R"	"b"	"READ"					
(Gráf ico)							

El almacenamiento de la información en este caso es exactamente igual al anterior. Se emplean 5 bytes, uno es el exponente y el resto la mantisa. Estos 5 bytes se encuentran siempre tras los bytes que denominan a la variable, que en este caso son tres.

Sin embargo, sólo uno de los tres bytes concuerda en el código con su carácter correspondiente, el resto no. El ordenador debe identificar el primer y último byte del nombre, y para ello utiliza otros bits discriminadores. El primer byte se diferencia de los demás en que sus tres últimos bits (bits 5, 6 y 7) tienen el valor 1 0 1. Si el ordenador encuentra un byte que no sea de dato y tiene ese valor en sus tres últimos bits, sabe que se trata de una variable numérica de más de una letra. Si se utilizara los mismos bits identificadores que en caso anterior, confundiría los bytes siguientes con bytes de dato, dando un valor erróneo a la variable.

Tan solo debe saber identificar al último código del nombre, pues a partir de él se encontrarán los 5 bytes de dato. Todos los datos que se encuentran entre el primer y último byte deben ser interpretados como el resto de los caracteres del nombre de la variable. El último byte lo identifica de los demás por su último bit (bit 7) , que es igual a 1. En el ejemplo, el último byte es igual a 227 d / 11100011 b. Si reemplazamos el 7° bit por un 0, obtendremos como resultado el valor 99 d /01100011 b. Ya este código corresponde el carácter de "c", que es el último del nombre de la variable " ABC".

Los bytes que estén entre el primero y el último almacenan el resto de los caracteres y sus bites identificadores tienen la misma forma que en el caso de variable numérica con letra única (011). Observa que el carácter correspondiente al segundo código es el "b", que también es la segunda letra de la variable.

Cuando el ordenador es requerido para imprimir el contenido de la variable " ABC", busca primero los tres bits identificado res (101). Seguidamente debe encontrar el último byte que define el nombre, el su 7° bit es igual a 1. Si la longitud del nombre coincide con el número de bytes, entonces es cuando compara los bits 0 a 4 de los códigos de los caracteres de la variable con los de los bytes que almacenan el nombre en memoria. Si estos son iguales, la variable ha sido hallada.

(3) VARIABLE ALFANUMERICA : A$, CONTENIDO "Bit"

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
65 d 01000001 b Caracter "A"	3	0	66 "B"	105 "i"	116 "t"

Seguidamente voy a explicar el método que utiliza el ordenador para almacenar variables alfanuméricas, también llamadas cadenas. Como supongo que sabréis, en el ZX Spectrum sólo puede haber 26 variables alfanuméricas, pues se denominan únicamente con una letra y el alfabeto que utiliza el ordenador consta de ese mismo número de caracteres.

El ordenador almacena el código de la letra de la variable con sus tres últimos bits con valor 010. Estos son los identificadores para las cadenas. El resto de los bits forman el nombre de la

variable. La elección de los indicadores 010 tiene como consecuencia que el byte completo, que contiene el nombre de la cadena, es el código correspondiente a su nombre en mayúsculas. En el ejemplo, la cadena se denomina "A$", que concuerda con el código almacenado en memoria. Solamente se dispone de este método para almacenar variables alfanuméricas en memoria, de manera que éstas constan de una única letra.

La longitud de las cadenas está sólo limitada por la memoria libre del ordenador. Ninguna cadena puede pasar la longitud 41473, al ser ésta la máxima capacidad de memoria libre que un 48K puede ofrecer. El ordenador reserva dos direcciones de memoria para la longitud de la cadena. Aqu í se almacenarán, en forma de byte menos significativo -byte más significativo el número de caracteres que contiene la variable. En el ejemplo, son sólo 3 los caracteres, de manera que el primer byte de los que definen la longitud contiene el valor 3. El segundo contiene el valor 0.

A partir del cuarto byte se almacena el contenido de la cadena.

Cada dirección de memoria se ocupa con el código correspondiente al carácter de la variable. Se ocuparán tantas direcciones de memoria como caracteres tenga la cadena. Es indiferente que la cadena contenga letras, cifras o s ímbolos. Cada uno de ellos tiene su código correspondiente en el juego de caracteres y puede ser expresado con un byte. En el ejemplo vemos que el cuarto byte almacena el valor 66d que es el código de la letra "B", primer carácter del contenido de la cadena A$. El segundo y tercer byte contienen los códigos 105 d y 116 d, que corresponden repectivamente a los caracteres "i" y "t".

La longitud definida por los bytes 2 y 3 indica al ordenador el número de bytes que debe interpretar como caracteres.

(4) VARIABLE DE CONTROL DE BUCLE FOR-NEXT : F

Byte 1	Byte 2-6	Byte 7-11	Byte 12-16	Byte 17-18	Byte 19
230 d 11100110 Carácter "NEW"	VALOR	LIMITE	PASO	NUMERO DE LINEA	NUMERO DE ORDEN

Las variables de control tienen la función de almacenar una serie de valores para un bucle. La orden NEXT modifica el contenido de estas variables, incrementándolas o decrementándolas. Primero veamos cómo es identificada una variable de este tipo. Los bits discriminadores son, como de costumbre, los tres últimos, los números 5, 6 y 7. Estos contienen siempre el valor 111. El resto de los bits almacenan el nombre de la variable. Si reemplazamos los bits identificadores por el valor 011, obtendremos el código correspondiente al carácter en minúscula del nombre. En el ejemplo, la variable se llama F. En memoria aparece el byte 11100110 b y tras suprimir los bits 5, 6 y 7 el byte cambia a 01100110 b / 120 d, que no es más que el código del carácter de "f".

A diferencia de las variables numéricas, las de control sólo pueden contener una letra. La razón estriba en que se reserva sólo un byte para el nombre.

Una variable de control se inicializa de la siquiente manera:

10 FOR F = 1 TO 1000 STEP 10

Su valor inicial es de 1 y se irá incrementando en pasos de 10 hasta que llegue a sobrepasar el valor 1000 d. Esta operación la realiza el comando NEXT. Para cada uno de estos valores se reservan 5 bytes. En primer lugar (bytes 2 a 6) se encuentra el valor que contenga la variable, almacenando en coma flotante (4 bytes de mantisa y uno de exponente). El contenido de estos 5 bytes será modificado por la orden NEXT .

Los 5 bytes siguientes (bytes 7 a 11) contienen el límite que el bucle puede alcanzar, en este caso 1000d. El sistema utilizado es el mismo que anteriormente, la coma flotante.

Otros 5 bytes se reservan para guardar el paso que se haya elegido para el bucle. Si el programador no inicializa el paso, éste se pone automáticamente a uno. En el ejemplo, estos 5 bytes contendrían en el sistema de coma flotante el valor 10d.

Aparte de estos datos, el ordenador almacena el número de línea del bucle. Si no se ha llegado al límite especificado, el comando NEXT ejecuta un salto al número de línea indicada por dos bytes (bytes 17 y 18) . El número de línea se almacena en forma byte menos significativo, byte más significativo.

Dado que el ZX Spectrum permite incluir más de una orden en una línea. El número de orden debe ser también especificado. Un solo byte almacena este valor (byte 19). En el ejemplo, los bytes 17 y 18 contendrían el valor 10 y 0 y el byte 19 el valor 2. Esto indica que el comando NEXT debe ejecutar un salto a la línea 10, orden número dos. Dado que no existe una orden número dos en la I ínea 10, el ordenador pasa inmediatamente a ejecutar la orden de la próxima línea.

Para almacenar una variable de control se precisan por tanto 19 bytes en el área de variables, a los que hay que sumar los utilizados por el área BASIC.

(5) MATRIZ NUMERICA : A (2,2)

Byte 1	Byte 2-3	Byte 4	Byte 5-6	Byte 7-8	Byte 9-13	Byte 14-18
129 d 10000001	25 0	2	2 0	2 0	(1,1)	(1,2)

Byte 19-23	Byte 24-28
(2,1)	(2,2)

Las matrices numéricas tienen la función de almacenar un conjunto de números en una matriz de n dimensiones, donde n puede ser de 1 a 255. Al igual que las variables de controlo las cade-

nas, se ha reservado únicamente 1 byte para el nombre de la matriz. Los tres bits identificadores tienen el valor 100. El resto de ellos contienen el nombre de la matriz. En el ejemplo, el primer byte, que es el que siempre almacena el nombre, tiene el valor 10000001 b /129 d. Reemplazando los bits identificadores por los bits 011, obtendremos el código del carácter del nombre de la variable (01100001 b = 97 d = CODE "a").

Lo verdaderamente importante son los bytes que vienen a continuación y que almacenan vital información. Los bytes 2 y 3 guardan la longitud utilizada por los datos que le siguen y los elementos de la matriz.

El cuarto byte almacena el número de dimensiones de la matriz. En el ejemplo, la matriz es bidimensional, de manera que este byte contiene el valor 2. El número máximo de dimensiones es por ello de 255, el dato mayor que se puede expresar con un byte.

Para cada una de las dimensiones se reservan dos bytes. Se comienza siempre a definir el valor de la primera dimensión que en el ejemplo es igual a 2. Los bytes 5 y 6 almacenan un 2 y un 0 respectivamente. A continuación se guarda el valor de la segunda dimensión, y así hasta la última. Si solamente existe una dimensión se utilizarán únicamente dos bytes, el quinto y sexto.

Una vez definida la última dimensión se pasa a almacenar el valor de cada elemento de la matriz. Para cada uno de ellos se utilizan 5 bytes, pues estos números se almacenan en el sistema de coma flotante. Si la matriz tiene 4 números, serán 20 los bytes utilizados sólo por los elementos. Las matrices tridimensionales o cuatridimensionales ocupan mucha más memoria. A la hora de programar es bueno saber este tipo de detalles, para evitar que la memoria se consuma con facilidad.

Los elementos de la matriz se ordenan de una forma determinada. En el ejemplo lo podrás ver claro: en primer lugar viene el elemento con dimensiones (1, 1) , el primero de la matriz. A continuación aumenta el valor de la segunda dimensión y se almacena el (1,2). Una vez acabada con la segunda dimensión, el valor de la primera aumenta; el tercer elemento es el (2,1). El último elemento de la matriz es siempre el que contiene los valores máximos de cada dimensión, en este caso, el (2,2). El valor de cada elemento es el inicio igual a cero.

Ahora podemos entender mejor la función de los bytes 2 y 3. Estos contienen la longitud total de los elementos y las dimensiones más uno, que corresponde al dato de número de dimensiones. En el ejemplo, estos bytes contienen el valor 20 (elementos) + 4 (dimensiones) + 1 (número de dimensiones) = 25. Contiene el número de direcciones de memoria ocupadas por los datos a partir de él mismo hasta la siguiente variable.

(6) MATRIZ ALFANUMERICA : A$(2,2) A$(1)="AA" A$(2)="BB"

Byte 1	Byte 2-3	Byte 4	Byte 5-6	Byte 7-8	Byte 9-10	Byte 11-12
193 d						
11000001	9 0	2	2 0	2 0	65 65	66 66
Carácter					"A" "A"	"B" "B"
"STR$"						

Las matrices alfanuméricas tienen la función de almacenar caracteres en un conjunto de dimensiones definidas por el programador. Las matrices se crean con el comando DIM. Junto al nombre de la matriz se escriben, entre paréntesis, las dimensiones de la misma. Una matriz alfanumérica se identifica de entre las otras variables por sus tres bits identificadores, los bits 5, 6 y 7. Estos tienen el valor 110 y el resto de los bits son los que almacenan el nombre de la matriz. Reemplaza los últimos 3 bits por el valor 011 y comprobarás cómo el resultado coincide con el código del carácter que da nombre a la matriz.

La estructura de los datos de la matriz alfanumérica en memoria es muy parecida a la de la matriz numérica. Antes de almacenar los elementos, el contenido de la matriz, se guarda información relativa al conjunto de caracteres. Los bytes 2 y 3 almacenan el número de direcciones de memoria ocupadas por los elementos, las dimensiones y el byte que almacena el número de dimensiones. Estos bytes también se usaban en la matriz numérica para saber la dirección de memoria de la próxima variable.

La función de almacenar el número de dimensiones es la que cumple el cuarto byte. Su contenido, como en el caso anterior, no puede superar el valor 255 d.

El valor de cada dimensión está definido por 2 bytes, en forma byte menos significativo - byte más significativo. Estos bytes vienen a continuación del cuarto y hay grupos de 2 bytes como dimensiones tenga la matriz. En el ejemplo, la matriz es bidimensional. El cuarto byte contiene el valor 2d. Los bytes 5-6 almacenan el valor de la primera dimensión, que es igual a 2d. A partir de ésta, se suceden los valores de las otras dimensiones, hasta la última. Los bytes 7-8 contienen el valor de la segunda dimensión, también igual a 2 d. Si se crea una matriz unidimensional, la primera y última serán la misma dimensión y solamente se utilizarán dos bytes.

Los elementos se almacenan a partir de la definición de la última dimensión. En el ejemplo se creaba una matriz de dimensiones 2 x 2, lo que quiere decir, de dos elementos alfanuméricos de dos caracteres de longitud. Primero se almacena el código del primer carácter del primer elemento y después el del segundo carácter del mismo elemento. A continuación viene el código del primer carácter del segundo elemento y en el último lugar el código del segundo carácter del segundo elemento. Se sigue el mismo orden que en la matriz numérica: (1,1), (1,2), (2, 1) y (2, 2).

Para cada uno de estos códigos se utiliza una dirección de memoria. Las matrices alfanuméricas son mucho más económicas que las numéricas, en las que para cada elemento se necesitan 5 bytes. Al dimensionar una matriz de caracteres, todos los elementos reciben el valor 32 d, cuyo carácter correspondiente es el de espacio.

- Ya conocemos la forma de almacenar y estructurar la información del área de variables. El programa assembler de este capítulo tendrá la función de investigar esta zona de la memoria y de informar de las variables existentes, imprimiendo sus contenidos. El área de variables comienza en la dirección de memoria determinada por la variable del sistema VARS (5C4B h y 5C4C h / 23627 d y 23628 d). A medida que el programa BASIC va creciendo en tamaño, esta zona es "empujada" hacia arriba, ocupando direcciones de memoria superiores. El final del área de variables lo determina un "End-Marker", es decir un código especial que bajo determinadas circunstancias (como ésta) informa al sistema operativo del fin de la zona de variables. Este código es el 80h / 128 d.

Introduce el siguiente programa :

```
10 LET A = 0: LET ABC = 0
20 LET A$ = "Bit"
30 DIM A (2,2) : DIM A$ (2,2)
40 FOR F = PEEK 23627 + 256 * PEEK 23628 TO 66E3
50 PRINT F;TAB 10;PEEK F;TAB 20;CHR$ PEEK AND PEEK F > 32
60 NEXT F
```

Su única función es imprimir los códigos de la zona hasta más allá del final de la memoria (en algún momento aparecerá el informa "B-Integer ...". Tú eres el que debe analizar qué bytes corresponden a qué variables y comprobar que lo dicho anteriormente coincide con la realidad.

Este será un buen ejercicio antes de dar paso al programa principal, un programa en código máquina que analiza la zona de variables, fijándose en los bits indicadores para saber de qué tipo de variable se trata e imprimir toda la información que trae consigo.

Recibe el nombre "Listado de variables" y tiene una longitud de 509 bytes, casi medio K de puro código máquina. Consta de 7 apartados, cada uno de los cuales será analizado separadamente. A continuación se encuentra el listado assembler:

===

Especificaciones : "Listado de Variables" para ZX Spectrum 16K/48K.
Descripciòn General : Se listan todas aquellas variables que esten almacenadas en memoria con sus correspondientes contenidos.
Longitud : 509 bytes.
Entrada : Ningùn requerimiento.
Salida : Ningùn requerimiento.
Nota : Las direcciones de memoria del programa son las que pertenecen a la versiòn de 48K.Los usuarios del modelo de 16K deben restar 8000H / 32768 d. Al definirse las etiquetas, se encuentran dos tablas,una para los usuarios de 48K y otra para los de 16K.

```
FD5B          00100          ORG FD5B H        ;
FD5B  CD6B0D  00110          CALL CLS          ; BORRA PANTALLA
FD5E  3E02    00120          LD A,2 D          ; ABRE CANAL
FD60  CD0116  00130          CALL OPEN         ; NUMERO DOS
FD63  2A4B5C  00140          LD HL, (VARS)     ; HL = PUNTERO
FD66  7E      00150 SIG-VAR  LD A, (HL)        ; A = PEEK HL
FD67  FE80    00160          CP 128 D          ; ¿ A=128 D ?
FD69  C8      00170          RET Z             ; RET SI ES ASI
FD6A  CB7F    00180          BIT 7,A           ; TEST BIT 7
FD6C  2008    00190          JR NZ,BIT 7=1     ; SALTA SI BIT=l
```

```
FD6E   CB6F     00200                BIT 5,A             ; TEST BIT 5
FD70   C2FCFE   00210                JP NZ,VAR-A         ; BIT=l -> VAR-A
FD73   C3D9FE   00220                JP,VAR-A$           ; BIT=0 -> VAR-A$
FD76   CB77     00230  BIT 7=1       BIT 6,A             ; TEST BIT 6
FD78   2008     00240                JR NZ,BIT 6=1       ; SALTA SI BIT=l
FD7A   CB6F     00250                BIT 5,A             ; TEST BIT 5
FD7C   C2C9FE   00260                JP NZ,VAR-ABC       ; BIT=l-> VAR-ABC
FD7F   C36EFE   00270                JP,VAR-A(l)         ; BIT=0->VAR-A(1)
FD82   CB6F     00280  BIT 6=1       BIT 5,A             ; TEST BIT 5
FD84   C2E3FD   00290                JP NZ,VAR-BUC       ; BIT=l-> VAR-BUC
FD87   7E       00300  VAR-A$(1)     LD A, (HL)          ; A = PEEK HL
FF88   EE80     00310                XOR 128 D           ; 128D=10000000 B
FD8A   D7       00320                RST 10H             ; ¡ IMPRIMELO !
FD8B   3E24     00330                LD A,36 D           ; A = CODE "$"
FD8D   D7       00340                RST 10H             ; ¡ IMPRIMELO !
FD8E   23       00350                INC HL              ; SIGUIENTE BYTE
FD8F   5E       00360                LD E, (HL)          ; DE=NUMERO TOTAL
FD90   23       00370                INC HL              ; DE DIMENSIONES+
FD9l   56       00380                LD D, (HL)          ; ELEMENTOS + 1
FD92   D5       00390                PUSH DE             ; DE EN STACK
FF93   23       00400                INC HL              ; SIGUIENTE BYTE
FD94   46       00410                LD B, (HL)          ; B=NUMERO DIMEN.
FD95   C5       00420                PUSH BC             ; BC EN STACK
FD96   3E28     00430                LD A, 40 D          ; A = CODE " ( "
FD98   D7       00440                RST 10H             ; ¡ IMPRIMELO !
FD99   23       00450                INC HL              ; SIGUIENTE BYTE
FD9A   C5       00460  OTRADIM1      PUSH BC             ; BC EN STACK
FD9B   3E00     00470                LD A,0 D            ; REGISTROS A,E,B
FD9D   1E00     00480                LD E,0 D            ; IGUAL A CERO.
FD9F   56       00490                LD D, (HL)          ; REGISTROS D y C
FDA0   23       00500                INC HL              ; CONTIENEN LA
FDA1   4E       00510                LD C, (HL)          ; DIMENSION
FDA2   0600     00520                LD B,0 D            ;
FDA4   23       00530                INC HL              ; SIGUIENTE BYTE
FDA5   E5       00540                PUSH HL             ; HL EN STACK
FDA6   CDB22A   00550                CALL 10930 D        ; ANALIZA E IMPRI
FDA9   CDE32D   00560                CALL 11747 D        ; -ME EL NUMERO
FDAC   El       00570                POP HL              ; RECUPERA HL
FDAD   Cl       00580                POP BC              ; RECUPERA BC
FDAE   78       00590                LD A,B              ; REGISTRO A = B
FDAF   FE01     00600                CP 01 H             ; ¿ES A=l? SI->
FDB1   2803     00610                JR Z, FINDIM1       ; NO IMPRIMAS " , "
FDB3   3E2C     00620                LD A,44 D           ; A = CODE ","
FDB5   D7       00630                RST 10H             ; ¡ IMPRIMELO !
FDB6   10E2     00640  FINDIMl       DJ NZ,OTRADIM1      ; OTRA DIMENSION
FDB8   3E29     00650                LD A, 41 D          ; A = CODE ") "
FDBA   D7       00660                RST 10H             ; ¡ IMPRIMELO !
```

```
FDBB   2B         00670               DEC HL              ; HL APUNTA A UL-
FDBC   2B         00680               DEC HL              ; TIMA DIMENSION
FDBD   4E         00690               LD C, (HL)          ; PASA EL VALOR
FDBE   23         06700               INC HL              ; AL REGISTRO
FDBF   46         00710               LD B, (HL)          ; BC
FDC0   23         00720               INC HL              ; SIGUIENTE BYTE
FDC1   ED43B05C   00730               LD (23728) ,BC      ; BC EN 5CB0 H
FDC5   C1         00740               POP BC              ; RECUPERA N.DIM.
FDC6   D1         00750               POP DE              ; RECUPERA LONG.
FDC7   1B         00760               DEC DE              ; RESTA 1
FDC8   1B         00770   ENC-LONl    DEC DE              ; RESTA 2 PARA
FDC9   1B         00780               DEC DE              ; CADA DIMENSION
FDCA   10FC       00790               DJ NZ,ENC-LON1      ; HASTA QUE B = 0
FDCC   ED4BB05C   00800   SIG-ELE     LD BC, (23728)      ; RECUPERA BC
FDD0   3E0D       00810               LD A,13 D           ; A = CODE"ENTER"
FDD2   D7         00820               RST 10H             ; ¡ IMPRIMELO !
FDD3   7A         00830   CONT-ELE    LD A,D              ; REGISTRO A = D
FDD4   B3         00840               OR E                ; OPERACION OR
FDD5   1B         00850               DEC DE              ; DE = DE - 1
FDD6   CA66FD     00860               JP Z,SIG-VAR        ; DE=0 -> SIG-VAR
FDD9   7E         00870               LD A, (HL)          ; A = PEEK HL
FDDA   23         00880               INC HL              ; SIGUIENTE BYTE
FDDB   D7         00890               RST 10H             ; ¡ IMPRIMELO !
FDDC   0B         00900               DEC BC              ; DECREMENTA BC
FDDD   78         00910               LD A,B              ; ¿ ES BC = 0 ?
FDDE   B1         00920               OR C                ; SI BC <> 0 ->
FDDF   20F2       00930               JR NZ,CONT-ELE      ; CONT-ELE
FDE1   18E9       00940               JR,SIG-ELE          ; BC=0-> SIG-ELE
FDE3   3E14       00950   VAR-BUC     LD A,20 D           ; A=CODE"INVERSE"
FDE5   D7         00960               RST 10H             ; ¡ IMPRIMELO !
FDE6   3E01       00970               LD A,l D            ; MODO"INVERSE 1"
FDE8   D7         00980               RST 10H             ; ¡ IMPRIMELO !
FDE9   7E         00990               LD A, (HL)          ; A = PEEK HL
FDEA   EEA0       01000               XOR 160 D           ; 160D=10100000 B
FDEC   D7         01010               RST 10H             ; ¡ IMPRIMELO !
FDED   111CFF     01020               LD DE,MENS1         ; PRIMER MENSAJE
FDF0   010A00     01030               LD BC,10 D          ; LONGITUD = 10
FDF3   CD3C20     01040               CALL PR-STRING      ; ¡ IMPRIMELO !
FDF6   23         01050               INC HL              ; SIGUIENTE BYTE
FDF7   7E         01060               LD A, (HL)          ; PASA LOS
FDF8   23         01070               INC HL              ; VALORES DE
FDF9   5E         01080               LD E, (HL)          ; EXPONENTE Y
FDFA   23         01090               INC HL              ; MANTISA A LOS
FDFB   56         01100               LD D, (HL)          ; REGISTROS A,E,
FDFC   23         01110               INC HL              ; D,C Y B
FDFD   4E         01120               LD C, (HL)          ; ALMACENAN EL
FDFE   23         01130               INC HL              ; VALOR DE LA
```

```
FDFF    46      01140           LD B, (HL)          ; VARIABLE DE
FE00    23      01150           INC HL              ; CONTROL
FE01    E5      01160           PUSH HL             ; HL EN STACK
FE02    CDB22A          01170           CALL 10930          ; ANALIZA E IMPRI
FE05    CDE32D          01180           CALL 11747          ; -ME EL NUMERO
FE08    El      01190           POP HL              ; RECUPERA HL
FE09    1126FF          01200           LD DE,MENS2         ;           SEGUNDO
MENSAJE
FE0C    010B00          01210           LD BC, 11 D         ; LONGITUD = 11
FE0F    CD3C20          01220           CALL PR-STRING      ; ¡ IMPRIMELO !
FE12    7E      01230           LD A, (HL)          ; PASA LOS
FE13    23      01240           INC HL              ; VALORES DE
FE14    5E      01250           LD E, (HL)          ; EXPONENTE y
FE15    23      01260           INC HL              ; MANTISA A LOS
FE16    56      01270           LD D, (HL)          ; REGISTROS A,E,
FE17    23      01280           INC HL              ; D,C y B
FE18    4E      01290           LD C, (HL)          ; ALMACENAN EL
FE19    23      01300           INC HL              ; LIMITE DE LA
FEIA    46      01310           LD B, (HL)          ; VARIABLE DE
FE1B    23      01320           INC HL              ; CONTROL
FEIC    E5      01330           PUSH HL             ; HL EN STACK
FE1D    CDB22A          01340           CALL 10930 D        ; ANALIZA E IMPRI
FE20    CDE32D          01350           CALL 11747 D        ; -ME EL NUMERO
FE23    El      01360           POP HL              ; RECUPERA HL
FE24    1131FF          01370           LD DE,MENS3         ; TERCER MENSAJE
FE27    010900          01380           LD BC,9 D           ; LONGITUD = 9
FE2A    CD3C20          01390           CALL PR-STRING      ; ¡ IMPRIMELO !
FE2D    7E      01400           LD A, (HL)          ; PASA LOS
FE2E    23      01410           INC HL              ; VALORES DE
FE2F    5E      01420           LD E, (HL)          ; EXPONENTE y
FE30    23      01430           INC HL              ; MANTISA A LOS
FE31    56      01440           LD D, (HL)          ; REGISTROS A,E,
FE32    23      01450           INC HL              ; D, C Y B
FE33    4E      01460           LD C, (HL)          ; ALMACENAN EL
FE34    23      01470           INC HL              ; PASO DE LA
FE35    46      01480           LD B, (HL)          ; VARIABLE DE
FE36    23      01490           INC HL              ; CONTROL
FE37    E5      01500           PUSH HL             ; HL EN STACK
FE38    CDB22A          01510           CALL 10930 D        ; ANALIZA E IMPRI
FE3B    CDE32D          01520           CALL 11747 D        ; -ME EL NUMERO
FE3E    El      01530           POP HL              ; RECUPERA HL
FE3F    113AFF          01540           LD DE,MENS4         ;           CUARTO
MENSAJE
FE42    010F00          01550           LD BC,15 D          ; LONGITUD = 15
FE45    CD3C20          01560           CALL PR-STRING      ; ¡ IMPRIMELO !
FE48    3E00    01570           LD A,0              ; REGISTROS A, E y
FE4A    1E00    01580           LD E,0              ; B IGUAL A CERO.
FE4C    56      01590           LD D, (HL)          ; REGISTROS C y D
FE4D    23      01600           INC HL              ; CONTIENEN EL
```

```
FE4E  4E      01610           LD C, (HL)      ; NUMERO DE LINEA
FE4F  23      01620           INC HL          ; DEL BUCLE
FE50  0600    01630           LD B,0          ;
FE52  E5      01640           PUSH HL         ; HL EN STACK
FE53  CDB22A  01650           CALL 10930 D    ; ANALIZA E IMPRI
FE56  CDE32D  01660           CALL 11747 D    ; -ME EL NUMERO
FE59  El      01670           POP HL          ; RECUPERA HL
FE5A  1149FF  01680           LD DE,MENS5     ; QUINTO MENSAJE
FE5D  010F00  01690           LD BC,15 D      ; LONGITUD = 15
FE60  CD3C20  01700           CALL PR-STRING  ; ¡ IMPRIMELO !
FE63  7E      01710           LD A, (HL)      ; A = N.SENTENCIA
FE64  C630    01720           ADD A,48 d      ; SUMALE 48 D
FE66  D7      01730           RST 10H         ; ¡ IMPRIMELO !
FE67  23      01740           INC HL          ; SIGUIENTE BYTE
FE68  3E0D    01750           LD A,13 D       ; A = CODE"ENTER"
FE6A  D7      01760           RST l0H         ; ¡ IMPRIMELO !
FE6B  C366FD  01770           JP SIG-VAR      ; SIG. VARIABLE
FE6E  7E      01780 VAR-A(l)  LD A, (HL)      ; A = PEEK HL
FE6F  EEC0    01790           XOR 192 D       ; 192D=11000000 B
FE71  D7      01800           RST 10H         ; ¡ IMPRIMELO !
FE72  23      01810           INC HL          ; SIGUIENTE BYTE
FE73  5E      01820           LD E, (HL)      ; DE=NUMERO TOTAL
FE74  23      01830           INC HL          ; DE DIMENSIONES+
FE75  56      01840           LD D, (HL)      ; ELEMENTOS + 1
FE76  D5      01850           PUSH DE         ; DE EN STACK
FE77  23      01860           INC HL          ; SIGUIENTE BYTE
FE78  46      01870           LD B, (HL)      ; B=N.DIMENSIONES
FE79  C5      01880           PUSH BC         ; BC EN STACK
FE7A  3E28    01890           LD A,40 D       ; A = CODE " ("
FE7C  D7      01900           RST 10H         ; ¡ IMPRIMELO !
FE7D  23      01910           INC HL          ; SIGUIENTE BYTE
FE7E  C5      01920 OTRADIM2  PUSH BC         ; BC EN STACK
FE7F  3E00    01930           LD A,0 D        ; REGISTROS A, E Y
FE81  lE00    01940           LD E,0 D        ; B IGUAL A CERO
FE83  56      01950           LD D, (HL)      ; REGISTROS C y D
FE84  23      01960           INC HL          ; CONTIENEN LA
FE85  4E      01970           LD C, (HL)      ; DIMENSION
FE86  23      01980           INC HL          ; SIGUIENTE BYTE
FE87  0600    01990           LD B,0 D        ;
FE89  E5      02000           PUSH HL         ; HL EN STACK
FE8A  CDB22A  02010           CALL 10930 D    ; ANALIZA E IMPRI
FE8D  CDE32D  02020           CALL 11747 D    ; -ME EL NUMERO
FE90  El      02030           POP HL          ; RECUPERA HL
FE91  C1      02040           POP BC          ; RECUPERA BC
FE92  78      02050           LD A,B          ; REGISTRO A = B
FE93  FE01    02060           CP 01H          ; ¿ES A = l?SI->
FE95  2803    02070           JR Z,FINDIM2    ; NO IMPRIMAS ", "
```

FE97	3E2C	02080		LD A,44 D	; A = CODE ","
FE99	D7	02090		RST 10H	; ¡ IMPRIMELO !
FE9A	10E2	02100	FINDIM2	DJ NZ,OTRADIM2	; OTRA DIMENSION
FE9C	3E29	02110		LD A,41 D	; A = CODE ")"
FE9E	D7	02120		RST 10H	; ¡ IMPRIMELO !
FE9F	C1	02130		POP BC	; RECUPERA N.DIM.
FEA0	D1	02140		POP DE	; RECUPERA LONG.
FEA1	1B	02150		DEC DE	; RESTA 1
FEA2	1B	02160	ENC-LON2	DEC DE	; RESTA 2 PARA
FEA3	1B	02170		DEC DE	; CADA DIMENSION
FEA4	10FC	02180		DJ NZ,ENC-LON2	; HASTA QUE B = 0
FEA6	3E0D	02190	SIG-NUM	LD A,13 D	; A = CODE"ENTER"
FEA8	D7	02200		RST 10H	; ¡ IMPRIMELO !
FEA9	7A	02210		LD A,D	; REGISTRO A = D
FEAA	B3	02220		OR E	; OPERACION OR
FEAB	1B	02230		DEC DE	; RESTA 5 PARA CA
FEAC	1B	02240		DEC DE	; -DA ELEMENTO DE
FEAD	1B	02250		DEC DE	; LA MATRIZ.UN NU
FEAE	1B	02260		DEC DE	; -MERO SE ALMACE
FEAF	1B	02270		DEC DE	; -NA CON 5 BYTES
FEB0	CA66FD	02280		JP Z,SIG-VAR	; HASTA QUE DE=0
FEB3	D5	02290		PUSH DE	; DE EN STACK
FEB4	7E	02300		LD A, (HL)	; PASA A LOS
FEB5	23	02310		INC HL	; REGISTROS A,E,D
FEB6	5E	02320		LD E, (HL)	; C Y B LOS
FEB7	23	02330		INC HL	; VALORES DEL
FEB8	56	02340		LD D, (HL)	; EXPONENTE Y LA
FEB9	23	02350		INC HL	; MANTISA
FEBA	4E	02360		LD C, (HL)	; CONTIENEN EL
FEBB	23	02370		INC HL	; VALOR DE 1
FEBC	46	02380		LD B, (HL)	; ELEMENTO DE LA
FEBD	23	02390		INC HL	; MATRIZ
FEBE	E5	02400		PUSH HL	; HL EN STACK
FEBF	CDB22A	02410		CALL 10930 D	; ANALIZA E IMPRI
FEC2	CDE32D	02420		CALL 11747 D	; -ME EL NUMERO
FEC5	E1	02430		POP HL	; RECUPERA HL
FEC6	Dl	02440		POP DE	; RECUPERA LONG.
FEC7	18DD	02450		JR,SIG-NUM	; SIG. NUMERO
FEC9	7E	02460	VAR-ABC	LD A, (HL)	; A = PEEK HL
FECA	EEE0	02470		XOR 224 D	; 224D=11100000 B
FECC	D7	02480	NO ULT	RST 10H	; ¡ IMPRIMELO !
FECD	23	02490		INC HL	; SIGUIENTE BYTE
FECE	7E	02500		LD A, (HL)	; A = PEEK HL
FECF	CB7F	02510		BIT 7,A	; TEST BIT 7
FED1	28F9	02520		JR Z,NO ULT	; BIT 7=0->NO ULT
FED3	CBBF	02530		RES 7,A	; BIT 7 = 0
FED5	D7	02540		RST 10H	; ¡ IMPRIMELO !

```
FED6    C300FF   02550             JP,NUMERICA      ; SALTA ADELANTE
FED9    7E       02560  VAR-A$     LD A, (HL)       ; A = PEEK HL
FEDA    D7       02570             RST 10H          ; ¡ IMPRIMELO !
FEDB    3E24     02580             LD A,36 D        ; A = CODE "$"
FEDD    D7       02590             RST 10H          ; ¡ IMPRIMELO !
FEDE    3E3D     02600             LD A,61 D        ; A = CODE "="
FEE0    D7       02610             RST 10H          ; ¡ IMPRIMELO !
FEEl    3E22     02620             LD A,34 D        ; A = CODE """
FEE3    D7       02630             RST 10H          ; ¡ IMPRIMELO !
FEE4    23       02640             INC HL           ; SIGUIENTE BYTE
FEE5    4E       02650             LD C, (HL)       ; BC CONTIENE EL
FEE6    23       02660             INC HL           ; NUMERO DE
FEE7    46       02670             LD B, (HL)       ; CARACTERES
FEE8    23       02680             INC HL           ; SIGUIENTE BYTE
FEE9    78       02690  SIG-CAR    LD A,B           ; REGISTRO A = B
FEEA    Bl       02700             OR C             ; OPERACION OR
FEEB    0B       02710             DEC BC           ; BC = BC - 1
FEEC    2805     02720             JR Z,FIN-CAR     ; BC=0 -> FIN
FEEE    7E       02730             LD A, (HL)       ; A = PEEK HL
FEEF    23       02740             INC HL           ; SIGUIENTE BYTE
FEF0    D7       02750             RST 10H          ; ¡ IMPRIMELO !
FEF1    18F6     02760             JR,SIG-CAR       ; SIGUIENTE CARAC
FEF3    3E22     02770  FIN-CAR    LD A,34          ; A = CODE """
FEF5    D7       02780             RST 10H          ; ¡ IMPRIMELO !
FEF6    3E0D     02790             LD A,13 D        ; A = CODE"ENTER"
FEF8    D7       02800             RST l0H          ; ¡ IMPRIMELO !
FEF9    C366FD   02810             JP,SIG-VAR       ; SIG.VARIABLE
FEFC    7E       02820  VAR-A      LD A, (HL)       ; A = PEEK HL
FEFD    CBAF     02830             RES 5,A          ; BIT 5 = 1
FEFF    D7       02840             RST l0H          ; ¡ IMPRIMELO !
FF00    3E3D     02850  NUMERICA   LD A,61 D        ; A = CODE "="
FF02    D7       02860             RST 10H          ; ¡ IMPRIMELO !
FF03    23       02870             INC HL           ; SIGUIENTE BYTE
FF04    7E       02880             LD A, (HL)       ; PASA A LOS
FF05    23       02890             INC HL           ; REGISTROS A, E,
FF06    5E       02900             LD E, (HL)       ; D, C Y B LOS
FF07    23       02910             INC HL           ; VALORES DEL
FF08    56       02920             LD D, (HL)       ; EXPONENTE y LA
FF09    23       02930             INC HL           ; MANTISA
FF0A    4E       02940             LD C, (HL)       ; CONTIENEN EL
FF0B    23       02950             INC HL           ; VALOR DEL
FF0C    46       02960             LD B, (HL)       ; NUMERO
FF0D    23       02970             INC HL           ; ALMACENADO
FF0E    E5       02980             PUSH HL          ; HL EN STACK
FF0F    CDB22A   02990             CALL 10930 D     ; ANALIZA E IMPRI
FF12    CDE32D   030G0             CALL 11747 D     ; -ME EL NUMERO
FF15    El       03010             POP HL           ; RECUPERA HL
```

```
FF16   3E0D     03020              LD A,l3 D            ; A = CODE"ENTER"
FF18   D7       03030              RST 10H              ; ¡ IMPRIMELO !
FF19   C366FD   03040              JP,SIG-VAR           ; SIG.VARIABLE
FFIC            03050    MENSl     DEFB 13,20,1         ; ' INVERSE 1
FFIF            03060              DEFM "Valor"         ;
FF24            03070              DEFB 20,0            ; INVERSE 0
FF26            03080    MENS2     DEFB 13,20,1         ; ' INVERSE 1
FF29            03090              DEFM "Limite"        ;
FF2F            03100              DEFB 20,0            ; INVERSE 0
FF31            03110    MENS3     DEFB 13,20,1         ; ' INVERSE 1
FF34            03120              DEFM "Paso"          ;
FF38            03130              DEFB 20,0            ; INVERSE 0
FF3A            03140    MENS4     DEFB 13,20,1         ; ' INVERSE 1
FF3D            03150              DEFM "N.de linea"    ;
FF47            03160              DEFB 20,0            ; INVERSE 0
FF49            03170    MENS5     DEFB 13,20,1         ; ' INVERSE 1
FF4C            03180              DEFM "N.de orden"    ;
FF56            03190              DEFB 20,0            ; INVERSE 0
0000            03200              END                  ;

0D6B   CLS
1601   OPEN
3C20   PR-STRING
```

ETIQUETAS VERSION 48 K

FD66	SIG-VAR
FD76	BIT 7=1
FD82	BIT 6=1
FD87	VAR A$(1)
FD9A	OTRADIM1
FDB6	FINDIM1
FDC8	ENC-LON1
FDCC	SIG-ELE
FDD3	CONT-ELE
FDE3	VAR-BUC
FE6E	VAR-A(1)
FE7E	OTRADIM2
FE9A	FINDIM2
FEA2	ENC-LON2
FE6A	SIG-NUM
FEC9	VAR-ABC
FECC	NO ULT
FED9	VAR-A$
FEE9	SIG-CAR
FEF3	FIN-CAR
FEFC	VAR-A

ETIQUETAS VERSION 16 K

7D66	SIG-VAR
7D76	BIT 7=1
7D82	BIT 6=1
7D87	VAR-A$(1)
7D9A	OTRADIM1
7DB6	FINDIM1
7DC8	ENC-LON1
7DCC	SIG-ELE
7DD3	CONT-ELE
7DE3	VAR-BUC
7E6E	VAR-A(1)
7E7E	OTRADIM2
7E9A	FINDIM2
7EA2	ENC-LON2
7E6A	SIG-NUM
7EC9	VAR-ABC
7ECC	NO ULT
7ED9	VAR-A$
7EE9	SIG-CAR
7EF3	FIN-CAR
7EFC	VAR-A

FF00	NUMERICA	7F00	NUMERICA
FF1C	MENS1	7F1C	MENS1
FF26	MENS2	7F26	MENS2
FF31	MENS3	7F31	MENS3
FF3A	MENS4	7F3A	MENS4
FF49	MENS5	7F49	MENS5

==

La estructura del programa "Listado de Variables" está formada por 7 apartados. El primero analiza el tipo de variable y accede a un segundo apartado donde se imprime su contenido. Existen 6 rutinas de esta clase, correspondientes a los 6 posibles tipos de variables. El apartado inicial se fija en los bits identificadores de las variables y ejecuta el salto adecuado a su estado. Seguidamente se encuentra una tabla resumen de las variables y sus bits identificadores:

Bits	Variab1e
011	Nurnérica de 1 letra
010	Cadena
101	Numérica de más de 1 letra
100	Matriz de números
111	Variab1e de control
110	Matriz de caracteres

Antes de comenzar a analizar cada uno de los apartados del programa, debo explicar dos nuevas entradas a rutinas ROM que aparecen en el listado:

(1) CALL 10930 d (2) CALL 11747 d

Las direcciones altas de la memoria ROM contienen rutinas de cálculo de expresiones, rutinas aritmáticas y la calculadora de coma flotante. La dirección 10930 d corresponde al comienzo de una rutina de cálculo de expresiones. La entrada a ella supone el almacenamiento en la pila del calculador de los valores de los registros A, B, C, D y E. La verdadera razón del uso de esta subrutina es que la segunda, ubicada a partir de la dirección 11747 d, afializa estos valores e imprime el resultado derivado de interpretar el registro A como el exponente y los registros E, D, C y B como los cuatro bytes de mantisa de un número escrito en coma flotante. Esta rutina forma parte de las aritméticas.

El uso de ambas nos permite transformar un número escrito en un sistema tan complejo como puede ser el de coma flotante en un número decimal que pueda ser comprendido por nosotros.

Recuerda que todas las variables que almacenan números (variable numérica de 1 ó más letras, variable de control y matriz de números) hacen uso de este método para guardar su contenido.

Aunque el número utilice solamente dos bytes (Ios números llamados "enteros", en el margen 0 a 65535 d), también pueden ser analizados e impreso. En este caso son otros los registros que se utilizan: el Acumulador, que contiene el valor del exponente, debe ser igual a cero. Los regis-

tros E y B deben ser también igual a cero. Los registro D y C deben contener respectivamente el byte menos significativo y el más significado. La entrada a las subrutinas son las mismas y en el mismo orden.

El primer apartado del programa es, como dijimos, el que analiza los bits identificadores y accede a una u otra rutina, dependiendo de su valor. Esta es la rutina principal del programa. Tiene su comienzo marcado en la dirección de memoria FD5B h (48K) / 7D5B h (16K) y ocupa las líneas 00100 a 00290 del listado.

La pantalla se borra y se abre el canal número dos para permitir la impresión de las variables y sus datos. El registro doble HL se inicializa con el contenido de la variable del sistema VARS. De esta manera, HL contiene la dirección de memoria del comienzo del área de variables. Este registro va a tener la misma función durante todo el programa, es el puntero de la zona de variables.

El contenido de la dirección de memoria expresada con el registro HL se introduce en el Acumulador. Se trata de una operación A = PEEK HL. El registro A contiene entonces el byte que define el tipo de variable y cuyos bits 5, 6 y 7 son los identificado res. En caso que el byte sea igual a 128 d, el área de variables ha llegado a su fin. Por ello se compara, antes de proseguir, el valor 128d con el contenido de A. Un retorno condicional se ejecutará si ambos números son iguales.

A partir de ahora debemos analizar los bits identificadores y acceder a la rutina correspondiente, dependiendo de su valor. Al inicio se realiza un test sobre el bit número 7 del Acumulador. Comprueba en la tabla anterior cómo solamente dos tipos de variables (variable numérica de 1 letra y cadena) tienen su séptimo bit con valor 0. Si no se trata de una variable de las mencionadas anteriormente, el indicador de cero tomará el valor 0, y se ejecutará un salto a la dirección de memoria expresada con la etiqueta "BIT 7 = 1 ", pues el bit 7 será igual a 1.

Ahora analizaremos el caso en que el bit 7 sea igual a 0. La única diferencia entre los bits identificadores de una variable numérica de 1 letra y una cadena es el bit 5. De manera que su estado se comprueba. Si es igual a 1, es señal inequívoca que el tipo de variable es el de numérica de letra única (bits 011).

En el caso contrario, que el bit 5 sea igual a 0, se trata de una variable alfanumérica (bits 010). En cada uno de estos dos casos se ejecuta un salto a la rutina VAR-A (Variable numérica de 1 letra -A-) o a la rutina VAR-A$ (Variable tipo Cadena) .

A continuación se analiza el caso en que el bit 7 es igual a 1. En este caso, se habrá ejecutado anteriormente un salto a la dirección de memoria expresada con la etiqueta "BIT 7 = 1 ". Existen cuatro casos posibles, correspondientes a los cuatro tipos de variables. Primeramente se separan los dos casos en que el bit 6 es igual a 1 de los otros dos casos en que el bit 6 es igual a 0. Se realiza un test sobre este bit y si es igual a 1, se ejecuta un salto a la dirección de memoria expresada con la etiqueta "BIT 6 = 1 ". Si el salto no se ha ejecutado, tenemos la certeza que los dos últimos bits son 1 (bit 7) y 0 (bit 6). Sólo hace falta comprobar el estado del quinto bit. Si como resultado de esta operación, se comprueba mediante el indicador de cero que el bit operación es igual a 1, los bits identificadores son 101 y corresponde a una variable numérica de más de una letra. Si

el quinto bit es igual a 0, se trata de una matriz de números (bits 100). En uno u otro caso se ejecuta un salto a la dirección de memoria expresada con la etiqueta "VAR-ABC" o "VAR-A(1)".

Nos encontramos ahora con el caso en que el bit 6 es igual a 1. Esto quiere decir que anteriormente se ejecutó un salto a la dirección de memoria etiquetada con "BIT 6 = 1 " y que ahora analizamos. Si se ha llegado hasta este punto del programa significa que el bit 7 y el bit 6 son iguales a 1. El bit 5 diferencia dos posibles casos: Variable de control y matriz de caracteres. Por ello se realiza un test sobre este bit. Si es igual a 1, se trata de una variable de control (bits 111) y se ejecuta un salto a la dirección de memoria "VAR-BUC", comienzo de la rutina de análisis de la variable de bucle.

No es necesario realizar un salto si el bit es igual a 0, correspondiente a la matriz de caracteres (bits 110) , pues la rutina comienza a continuación.

De esta manera se accede a la rutina correspondiente a cada tipo de variable, analizando cada uno de los bits identificadores y separando cada uno de los casos posibles. La rutina principal se accede una vez terminadas las rutinas que analizan las variables. El punto de entrada de esta rutina principal es el etiquetado con "SIG-VAR".

La rutina VAR-A$ (1)

Esta rutina se ocupa de analizar e imprimir el contenido de una matriz alfanumérica. Tiene su comienzo marcado por la etiqueta "VAR-A$ (1)", en la línea assembler 00300. HL contiene apunta hacia el área de variables. El contenido de la dirección de memoria que representa se introduce en el registro A y se realiza una operación XOR. La función de esto es modificar el último bit del Acumu lador , para que la instrucción RST 10H imprima el nombre de la variable en mayúsculas.

Imagina que la matriz se ha dimensionado como A$. El código correspondiente será 11000001 b, pues sus tres últimos bits son siempre 110. La operación XO R 128 d / 1000000 b modifica el byte de tal manera que se convierte en el código del carácter del nombre de la matriz en mayúsculas.

```
1   1   0   0   0   0   0   1                       =  193 d
             XOR
 1   0   0   0   0   0   0   0                      =  128 d
--------------------------------
0   1   0   0   0   0   0   1 b          =  65 d   (CODE "A")
```

El código resultante es impreso a continuación por la orden RST 10H. Este es uno de los muchos usos que XOR puede tener.

Tras el nombre se imprime el carácter de dólar ("$"), que indica al usuario que se trata de una variable que admite cualquier tipo de símbolos.

El registro HL se incrementa, conteniendo la dirección de memoria que almacena el número total de elementos y dimensiones más uno, para el número de dimensiones. Comprueba en el estudio anterior de las variables esta función del segundo y tercer byte. Este valor se introduce mediante direccionamiento indirecto por registro en el DE.

Para realizar esta operación, el registro doble HL debe incrementarse de nuevo entretanto. Nada más finalizarla por completo, el registro DE se pone a salvo guardándolo en el Stack. Más tarde lo utilizaremos. Por el mismo sistema se introduce en el registro B el número de dimensiones, valor que almacena el cuarto byte. También se guarda en la pila de máquina.

Las instrucciones que vienen a continuación tienen la función de analizar e imprimir todas las dimensiones de la matriz. El número de veces que deben repetirse es el contenido del registro B.

Antes de todo se imprime en pantalla el carácter de paréntesis abierto y se incrementa de nuevo el registro doble HL. Ahora contiene la dirección de memoria que guarda, en forma byte bajo-byte alto, el valor de la primera dimensión. El número de dimensiones vuelve a salvarse en el Stack. Esta parte está etiquetada con "OTRADIM1" y será utilizada de nuevo más adelante para analizar otra dimensión, si la hubiere.

Tal y como dijimos al principio, para analizar e imprimir un valor "entero" (de dos bytes) los registros A, E y B se cargan con el valor 0, mientras D y C contienen el byte bajo y el byte alto de la dimensión. El registro HL se incrementa de manera que contenga la dirección de memoria de la próxima dimensión, o en su defecto, del primer elemento. Para analizar el valor de la dimensión, se accede a las rutinas 10930 d y 11747 d. Mientras el valor de HL debe ser guardado en el Stack, para que no se pierda su importante contenido. Este registro es usado en ambas rutinas de la ROM para otros propósitos.

Seguidamente se recupera del Stack el resgistro BC. B contiene el número de dimensiones de la matriz. Si es diferente a uno, se imprimirá una coma en la pantalla, para separar una y otra dimensión. La instrucción DJ NZ, decrementa automáticamente el registro B, como señal de que una dimensión ya se ha procesado. Si B es diferente de cero, es decir, que existen aún más dimensiones, se procesa un salto relativo condicional negativo, a la dirección de memoria etiquetada con "OTRADIM1" ("Otra Dimensión").

La razón para comprobar anteriormente si el registro B es diferente o no a uno reside en la necesidad o no de imprimir la coma. Si B contiene el valor 1, quiere decir que era la última dimensión y que no hace falta imprimir una coma tras él. El programa continúa en la dirección de memoria expresada por la etiqueta "FINDIM1" ("Fin de Dimensiones"), saltándose las instrucciones que imprimen la coma.

Si este es el caso, el programa continúa imprimiendo el carácter de paréntesis cerrado. Todos los datos relativos al dimensionamiento de la matriz ya han sido impresos.

El registro HL se decrementa dos veces para que contenga la dirección de memoria de la última dimensión de la matriz. Este dato se introduce en el registro BC y se almacena momentáneamente en la dirección de memoria 5CB0 h /23728 d.

El registro HL se incrementa entretanto para que contenga de nuevo la dirección de memoria a partir de la cual se encuentran los elementos de- la matriz, si pero ¿cuántos?

Las instrucciones que vienen a continuación se encargan de calcular el número de elementos que consta la matriz. Por de pronto, los dos últimos valores del Stack se cargan en los registros BC y DE. El primero contiene el número de dimensiones de la matriz. El segundo almacena la longitud total de los elementos de la matriz, las dimensiones más uno, para el dato de número de dimensiones. DE contiene la longitud desde el tercer byte hasta el fin. El cálculo del número de elementos de la matriz se realiza restando de la longitud total los bytes necesarios para almacenar las dimensiones más uno, para el número de dimensiones.

La primera instrucción de decrementación se ocupa de restar a DE el dato del número de dimensiones. Las otras dos instrucciones DEC se repetirán B veces. De esta manera restamos 2 bytes por cada dimensión, pues este es el número de bytes necesarios para su almacenamiento y B almacena el número de dimensiones. La instrucción DJ NZ envía al programa a la parte etiquetada con "ENC-LON1 " ("Encuentra la Longitud 1 ").

Tras este proceso, que finaliza en la línea assembler 00800, DE contiene la longitud ocupada por los elementos de la matriz. Esto equivale al número de elementos de la matriz, pues se necesita 1 byte para cada uno de ellos, tal y como vimos al estudiar este tipo de variable. El registro BC se recupera de la dirección de memoria 5CB0 h / 23728 d y vuelve a contener el valor de última

dimensión. Estos dos registros son vitales para el proceso que comienza a continuación. Se trata de imprimir el contenido de la matriz en grupos. Se imprimirán tantos elementos juntos como la longitud de la última dimensión, de manera que si se dimensionó el conjunto A$ (20, 10), imprima 20 grupos de 10 caracteres cada uno y no 200 caracteres uno bajo el otro.

El registro HL sigue conteniendo la dirección de memoria del primer elemento de la matriz. Se imprime en pantalla el código 13, perteneciente al carácter de "ENTER", que obliga a imprimir el siguiente carácter una fila más abajo, en la columna cero.

A continuación se comprueba si DE es igual a cero. Este registro se va decrementando a medida que se repite el proceso por la instrucción de la línea 00850. Si es igual a cero, se habrá acabado de imprimir el contenido de la matriz y se debe proceder al análisis de la siguiente variable. Si se abandona esta rutina, el registro HL contiene la dirección de memoria que almacena el tipo y nombre de la próxima variable si la hubiera.

Si este no es el caso, se introduce en el Acumulador el contenido de la dirección de memoria que expresa ese registro doble. A contiene el código del carácter de un elemento de la matriz. Este será impreso por RST 10H. El registro HL se incrementa para que contenga la dirección de memoria del próximo elemento o si no existiera, de la próxima variable. Lo realmente importante de este proceso viene a continuación.

El registro BC, que almacena el valor de la última dimensión, se decrementa. Mediante la operación OR se comprueba si BC es o no es igual a cero. Si no es el caso, el proceso continúa en la dirección de memoria expresada por la etiqueta "CONT -ELE" ("Continúa con el Elemento") y el

próximo carácter se imprimirá junto al anterior. Pero si no es el caso, la dirección de memoria de destino se expresa con la etiqueta "SIG-ELE" ("Siguiente Elemento"), que vuelve a introducir en BC el valor de la última dimensión e imprime el carácter de "ENTER". De este modo, si se hubiera dimensionado una matriz A$(20, 10) , se imprimirían 10 caracteres uno junto al otro 20 veces. El primer parámetro se deriva del contenido de BC, la última dimensión y el segundo sería la consecuencia de ir decrementando DE cada vez que se imprime un elemento.

La rutina VAR-BUC:

Esta rutina tiene la función de analizar e imprimir el contenido de una variable de control, con todos los datos que la acompañan: límite máximo, paso, número de línea del bucle y número de orden dentro de la línea.

Su comienzo está etiquetado con "VAR-BUC" ("Variable de Bucle"), en la línea assembler 00950. El nombre de la variable es impresa en modo I NVE RSE, con el carácter de control 20 d y el dato 1 d. El registro HL entra en esta rutina con la dirección de memoria que almacena el tipo y nombre de la variable de control.

El acumulador se carga con este valor y se realiza una operación XOR 160 d /10100000 b, que modifica el estado del byte al código del carácter del nombre de la variable en mayúsculas. Es un caso análogo al de la rutina VAR-A$(1). El código resultante se imprime con la instrucción RST 10H.

La estructura de una variable de control indica que los 5 bytes siguientes al nombre contienen en forma de coma flotante el valor de la variable. Antes de que aparezca el valor en sí y para que el usuario tenga una referencia de ese número se imprime el mensaje "Valor". Para ello se hace uso de la rutina de ROM PR-STRING. El registro DE contiene la dirección de memoria donde están almacenados los códigos que forman los datos y el BC la longitud del mensaje. El primer valor se expresa mediante la etiqueta "MENS1 " ("Mensaje 1 ") y el segundo es igual a 10d.

Tras el CALL, el registro HL se incrementa para que contenga la dirección de memoria a partir de la cual se almacenan los 5 bytes de dato (1 de exponente y cuatro de mantissa). Estos valores se introducen en los registros A, E, D, C y B, en este orden. Por supuesto, el registro HL debe ser incrementado cada vez. Las transferencias de valores a los registros se realizan mediante direccionamiento indirecto por registro.

El análisis y la impresión del valor de la variable de control se realiza mediante las rutinas de ROM ubicadas en las direcciones 10930 d y 11747 d y ya mencionadas. Antes de acceder a ellas, el contenido de HL debe ser guardado en Stack, para que no se pierda.

La. operación se repite con los próximos 5 bytes. Esta vez contienen, en forma de coma flotante, el valor del límite del bucle. Antes de todo, se imprime el mensaje apropiado, cuyos 11 d códigos están almacenados a partir de la etiqueta MENS2 ("Mensaje2") y que forman la palabra "Límite", en modo inverso y una fila más abajo. Los valores de los 5 bytes se introducen por el mismo sistema que antes en los registros A, E, D, C y B. Las rutinas de ROM almacenadas a par-

tir de las direcciones de memoria 10930 d y 11747 d cumplen la función de analizar e imprimir el número correspondiente.

La misma operación se repite por tercera vez consecutiva y para los 5 bytes siguientes. Esta vez, éstos almacenan el paso utilizado para aumentar el valor de la variable en cada NEXT. El mensaje que se imprime, también por el mismo método que antes, especifica el número que aparece a continuación. El registro HL va incrementándose e introduciendo en los registros A, E, D, C y B los valores del exponente y la mantissa. Se vuelven a utilizar las rutinas de las direcciones 10930 d y 11747 d.

Al terminar este proceso, el registro doble HL apunta hacia la dirección de memoria que contiene el byte bajo del número de línea del bucle. El byte alto lo ocupa la dirección de memoria siguiente. Antes de analizar este valor se imprime otro mensaje, esta vez indicando que se trata del número de línea donde se encuentra el comando FOR. Se vuelve a utilizar la rutina PR-STRING, los registros DE y BC.

Para analizar e imprimir este número de dos bytes el registro D debe contener el byte bajo y el C el byte alto del llamado número "entero". Los registros A, E y B deben contener el valor 0. Entonces entran en acción las rutinas ROM ubicadas en las direcciones 10930 d y 11747 d y que tanto hemos utilizado ya.

El registro HL finaliza apuntando al último dato de que dispone una variable de control. Al igual que para el resto de los datos, un mensaje apropiado se imprime en pantalla indicando que el número que viene a continuación es el número de orden siguiente al comando FOR dentro de la misma línea. Si el FOR se encuentra en primer lugar, este dato será igual a 2. Indica a la instrucción NEXT que debe dirigirse a la orden segunda de la línea especificada por los dos bytes anteriores. Para transformar el dato numérico en el código correspondiente al carácter de ese dato numérico, se le suma la constante 48 d. Si el dato es igual a 2, al sumarle 48 d se convertirá en 50 d, que es el código del carácter del número dos. Este sistema funciona sólo si el dato es menor o igual a 9.

En el acumulador se introduce el número de orden siguiente al FOR dentro de la misma línea. A este valor se le suma la constante 50d y se convierte en el código del carácter de aquel número, que puede ser impreso por la instrucción RST 10H.

Antes de abandonar la rutina VAR-BUC el registro doble HL se incrementa para que apunte a la dirección de memoria de la próxima variable. El código de "ENTER" también es impreso para acceder a la siguiente fila, columna cero. La rutina se abandona en la línea 01770, saltando a la parte denominada "SIG-VAR", en el comienzo del programa.

VAR-A (1):

Esta rutina comienza en la dirección de memoria especificada por la etiqueta "VAR-A(1)", en la línea assembler 01780. Como podréis comprobar, es muy parecida a la rutina VAR-A$(1); el ordenador tiene sistemas muy parecidos para el almacenamiento de matrices. Primeramente se in-

troduce en el Acumulador el byte que especifica el tipo de variable y la da nombre. Se realiza una operación XOR, con el único propósito de modificar los bits identificadores y que, como en los casos anteriores, el resultado sea el código del carácter del nombre de la matriz en mayúsculas. Este es impreso por la instrucción RST 10H que aparece a continuación.

Los bytes segundo y tercero tienen la función de almacenar la longitud compuesta por todos los elementos, las dimensiones, más uno, para el dato de los números de dimensiones. En la matriz de caracteres nos encontrábamos con el mismo caso.

Este valor es introducido mediante direccionamiento indirecto por registro en el DE. El registro HL debe incrementarse cada vez, para acceder al byte siguiente en memoria. El contenido de DE se almacena en el Stack. Al finalizar este proceso, el registro HL apunta hacia la dirección de memoria que guarda el cuarto byte. El contenido de esta dirección pasa al registro B y después a la pila de máquina.

Las instrucciones que vienen a continuación tienen la función de imprimir las dimensiones de la matriz numérica. Antes de ello se imprime el carácter de paréntesis abierto, que indica al usuario que los datos que vengan a continuación pertenecen al dimensionamiento de la matriz.

El proceso para imprimir las dimensiones es exactamente el mismo que el utilizado en la matriz alfanumérica. El número de dimensiones se almacena en el Stack y se cargan los registros A, E y B con el valor 0. Por otra parte, los registros D y C contienen el byte bajo y alto del número entero que expresa la dimensión. Se hace uso, como siempre, de las rutinas ROM de las direcciones 10930 d y 11747 d para analizar e imprimir el valor de la dimensión.

El valor de HL debe ser guardado entretanto en la pila de máquina. Al igual que en la rutina VAR-A$(1), se imprime una coma entre dimensión y dimensión. El registro que almacena el número de dimensiones, el registro B se recupera del Stack y se compara por medio del Acumulador con el valor 1. Si B es diferente de uno, la coma debe ser impresa, pues quiere decir que existe otra dimensión más. Pero si en cambio, B es igual a 1, se acababa de procesar la última y no hace falta imprimir otra coma. El comienzo del proceso, que debe ser repetido para cada dimensión, está expresado por medio de la etiqueta "OTRADIM2".

El registro doble se incrementa cada vez en dos y termina apuntando a la dirección de memoria a partir de la cual se almacenan los elementos de la matriz. Una vez se hayan acabado con todas las dimensiones, se imprime el carácter de paréntesis cerrado y se comienza otro proceso: el que calcula la longitud ocupada por los elementos de que consta la matriz.

Estos elementos son los que deben ser impresos. Al ser números, se necesitan 5 bytes para cada elemento, pues se almacenan en forma de coma flotante. El método para calcular su número se deriva del usado con las matrices alfanuméricas. Los registros BC y DE se recuperan del Stack. El primero contiene el número de dimensiones de la matriz, en el registro B. El segundo contiene la longitud compuesta por los elementos, las dimensiones, más uno para el número de dimensiones. La primera instrucción DEC resta uno en concepto de este último valor, el número de dimensiones.

Después se establece un proceso repetitivo que decrementa en dos el registro DE y lo hace B veces. De esta manera se resta de DE lo ocupado por las dimensiones, pues cada una necesita dos bytes, y su número está contenido en el registro B. Al finalizar este proceso, el registro DE contiene la longitud ocupada por los elementos de la matriz (5 veces más que su número). Mientras tanto, el registro HL no ha sido utilizado y sigue apuntando a la dirección de memoria donde comienzan a almacenarse los números.

La rutina VAR-ABC:

Esta rutina analiza e imprime el valor de una variable numérica cuyo nombre está compuesto por más de una letra. Este es el caso de una variable imaginaria llamada " ABC". Tiene su comienzo fijado por la etiqueta del mismo nombre y en el listado assembler figura en la línea 02460.

Las pocas instrucciones que figuran en la rutina tienen la función de imprimir correctamente el nombre, pues consta de dos tipos de bytes identificadores: uno con el primer byte que define el nombre y otro con el último.

La rutina se incorpora en un punto determinado de la rutina VAR-A (variable numérica de una letra), pues la forma de almacenar el número es la misma: en coma flotante. El punto de entrada de la rutina VAR-A está definido por la etiqueta NUMERICA.

Primeramente se carga en el Acumulador el contenido de la dirección de memoria expresada por el registro HL. Este byte es el que almacena e1 tipo y la primera letra de la variable. Se realiza una operación XOR 224 d /11100000 b. Comprueba tú mismo que el byte resultante y que pasa a ser automáticamente el nuevo valor del Acumu lador, es igual al código de la primera letra en mayúsculas de cualquier variable de este tipo. Este byte es impreso en pantalla.

La última letra de la variable se identifica de las demás, porque su último bit (el bit 7) es igual a uno. Los bytes entre la primera y la última letra forman y almacenan el resto de las letras de la variable. El registro HL se incrementa para pasar a la siguiente dirección de memoria, que almacena. Para saber si se está procesando la última letra de la variable y pasar a analizar el número, se comprueba el estado del séptimo bit. Si es igual a cero, se trata una letra que no es la última yel programa retrocede 7 direcciones de memoria, hasta la instrucción etiquetada con "NO ULT" ("No Ultima letra"). Se ejecuta una instrucción RST 10H, que imprime la letra y pasa a analizarse el siguiente byte.

Este proceso continuará hasta que se de con la última letra, cuyo byte tiene el séptimo bit con valor 1. En este caso no se ejecuta ningún salto. Este séptimo bit es puesto a cero con una instrucción RESET . Ahora ya representa el código de una letra y puede ser impreso por RST 10H.

La rutina continúa en la que analiza la variable numérica de una letra, cuando se procesan los 5 bytes de dato. Pero de esto ya hablaremos más adelante.

VAR-A$:

La rutina, cuyo comienzo está definido por la etiqueta con nombre "VAR-A$", tiene la función de imprimir el contenido de una cadena de caracteres de longitud sólo limitada por la memoria. En el listado assembler se encuentra en la línea 02560.

Como en las otras rutinas, lo primero que se realiza es la impresión del nombre de la variable. Afortunadamente no hay que ejecutar ninguna operación XOR, pues el código utilizado por el ordenador para identificar el tipo y el nombre de la variable es el mismo que el código de las letras mayúsculas. Una cadena con nombre A$, tendrá como código el 01000001 b / 65 d y este mismo número coincide con el código del carácter de la A mayúscula.

Para indicar al usuario que se trata de una cadena, se imprimen a continuación los caracteres de dólar ("$"), del signo de igual ("=") y el de comillas (" " ").

A continuación del byte del nombre de la variable se encuentran dos bytes que expresan la longitud de la cadena en forma byte menos significativo - byte más significativo. El registro HL se incrementa para que apunte al byte bajo y su contenido se introduce en el registro C. HL vuelve a incrementarse y el byte alto pasa al registro B. De esta manera, el registro doble BC contiene la longitud de la variable.

A partir del siguiente byte comienzan a almacenarse los códigos de los caracteres que forman la cadena. El registro HL se incrementa y apunta a esta dirección de memoria.

Las siguientes instrucciones forman un bucle que imprime cada uno de los caracteres de la cadena. El número de ellos se expresa por medio del registro doble BC. Se comprueba si es diferente a cero, y si este es el caso, se salta a la parte etiquetada con "FIN-CAR" ("Fin de Caracteres"). Pero si no es el caso, el código correspondiente al carácter es introducido en el Acumulador e impreso en la pantalla. El registro HL se incrementa entretanto. Este registro siempre contiene la dirección de memoria del próximo dato o de la próxima variable. El programa salta a la parte denominada "SIG-CAR" ("Siguiente Carácter"), donde se vuelve a comprobar si el registro BC, anteriormente decrementado, es igualo diferente a cero.

Si se ha terminado de imprimir todo el contenido de la cadena, ésta se cierra con un carácter de comillas ("), que indica al usuario el fin de la variable alfanumérica. El código 13 d, de "ENTER" coloca las coordenadas de impresión una fila más abajo, en la columna cero. El programa salta a la rutina principal, donde se analizará la siguiente variable, o en su defecto, se retornará al BASIC.

VAR-A:

Esta es la última rutina del programa "Listado de variables". Su función es, por una parte, el proceso de las variables numéricas de una letra, y por la otra, el proceso del contenido de las

variables numéricas de más de una letra. Ambos tipos de variable almacenan los números exactamente de la misma forma: en coma flotante, de manera que no sería lógico escribir dos rutinas exactamente iguales, si no hacer una participe de la otra.

El comienzo de la rutina está marcado por la etiqueta "VAR-A ", en la línea assembler 02820.

El nombre de la variable es introducido en el Acumulador. Para modificar el byte y convertirlo en el código del nombre de la variable, en mayúsculas se procesa la operación RES 5, A pues es el bit número cinco el único que se diferencia entre el código del nombre de la variable en mayúsculas y el byte identificador de la variable. También sería posible una operación XOR 32d / 00100000 b, pero he preferido la primera por introducir algo nuevo. El nombre de la variable, en mayúsculas, es impreso a continuación. Un signo de igual ("=") se imprimirá tras el nombre. Este es el punto de entrada de la rutina VAR-ABC. A partir de aquí ambas rutinas son equivalentes.

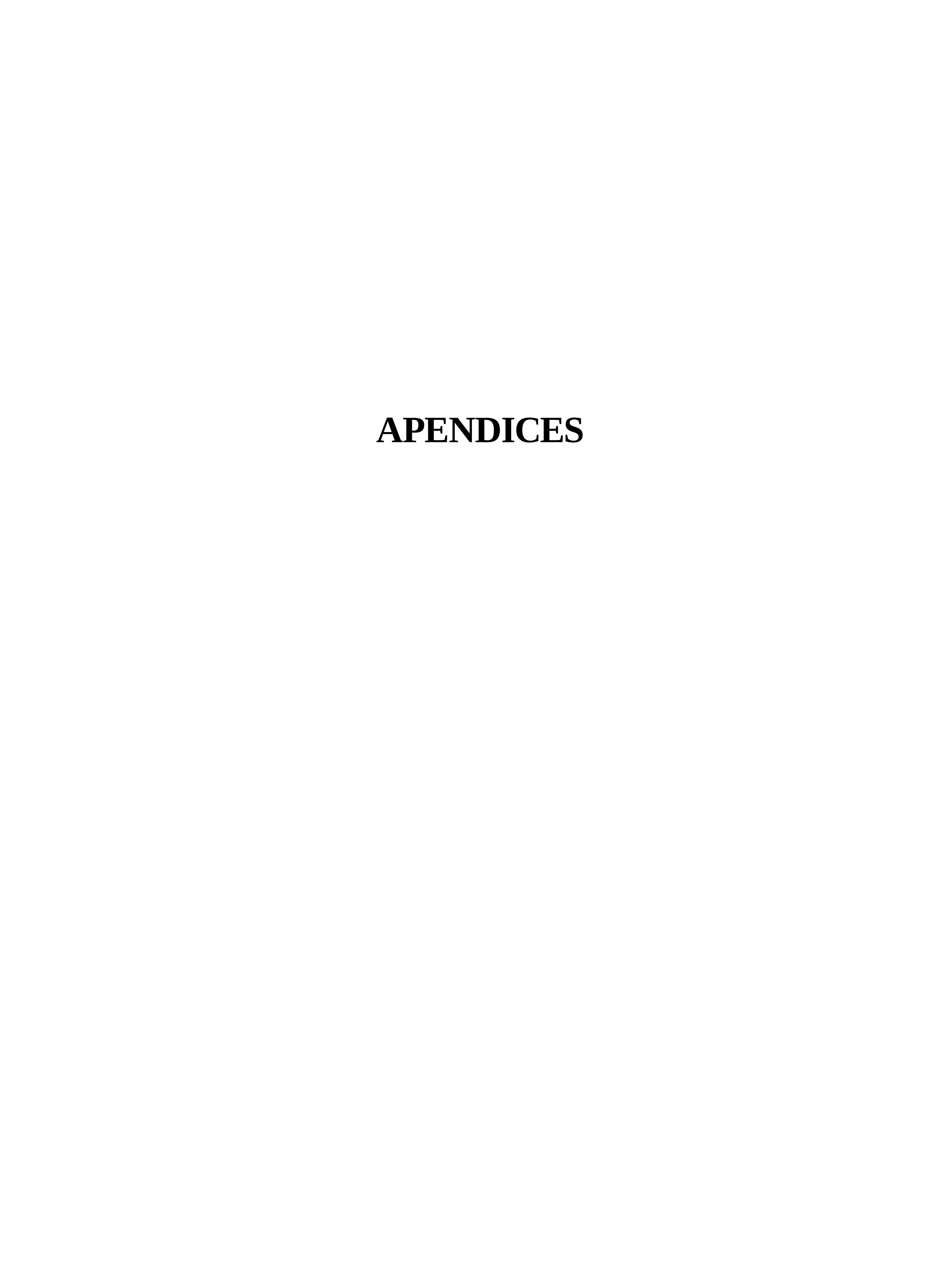
Los próximos cinco bytes almacenan el contenido de la variable numérica. Por medio de direccionamiento indirecto por registro y con el HL, se introduce en el Acumulador el valor del exponente y en los registros E, D, C y B los valores de la mantissa. HL debe incrementarse para pasar al siguiente byte. Se hace uso de las rutinas ROM 10930 d y 11747 d, que analizan e imprimen el número en pantalla. El registro HL debe ser salvado en el Stack mientras tanto. La rutina ya ha llegado a su fin. Se imprime el carácter de "ENTER" y se retorna al inicio del programa. Como siempre, HL contiene la dirección de memoria de la próxima variable si la hubiere.

Tras esta rutina aparecen en el listado los 5 diferentes mensajes utilizados para la rutina de variable de control. Las etiquetas indican el valor que DE debe tener al acceder a la rutina ROM PR-STRING.

```
 10 REM *** LISTADO DE VARIABLES ***
 20 REM Juan Mtz.Velarde,1984
 30 GO SUB 2E3: CLEAR CL
 40 GO SUB 2E3
 50 LET T=0
 60 FOR F=L1 TO L2
 70 READ A: POKE F,A
 80 LET T=T+A: NEXT F
 90 IF T<>53965 THEN PRINT "ERROR EN DATAS": STOP
100 DATA 205 , 107 , 13 , 62 , 2 , 205 , 1 , 22 , 42 , 75 , 92 , 126 , 254 , 128 , 200 ,
    203 , 127 , 32 , 8 , 203 , 111 , 194 , 252 , 254 , 195 , 217 , 254 , 203 , 119 , 32 ,
    8 , 203 , 111 , 194 , 201 , 254 , 195 , 110 , 254 , 203 , 111 , 194 , 227 , 253
200 DATA 126 , 238 , 128 , 215 , 62 , 36 , 215 , 35 , 94 , 35 , 86 , 213 , 35 , 70 , 197 ,
     62 , 40 , 215 , 35 , 197 , 62 , 0 , 30 , 0 , 86 , 35 , 78 , 6 , 0 , 35 , 229 , 205 , 178 ,
     42 , 205 , 227 , 45 , 225 , 193 , 120 , 254 , 1 , 40 , 3 , 62 , 44 , 215 , 16 , 226 , 62 ,
     41 , 215
250 DATA 43 , 43 , 78 , 35 , 70 , 35 , 237 , 67 , 176 , 92 , 193 , 209 , 27 , 27 , 27 , 16 ,
```

```
    252 , 237 ,75 , 176 , 92 , 62 , 13 , 215 , 122 , 179 , 27 , 202 , 102 , 253 , 126 ,
     35 , 215 , 11 , 120 , 177 , 32 , 242 , 24 , 233
300 DATA 62 , 20 , 215 , 62 , 1 , 215 , 126 , 238 , 160 , 215 , 17 , 28 , 255 , 1 , 10 ,
     0 , 205 , 60 , 32
320 DATA 35 , 126 , 35 , 94 , 35 , 86 , 35 , 78 , 35 , 70 , 35 , 229 , 205 , 178 , 42 ,
     205 , 227 , 45 , 225 , 17 , 38 , 255 , 1 , 11 , 0 , 205 , 60 , 32
340 DATA 126 , 35 , 94 , 35 , 86 , 35 , 78 , 35 , 70 , 35 , 229 , 205 , 178 , 42 , 205 ,
     227 , 45 , 225 , 17 , 49 , 255 , 1 , 9 , 0 , 205 , 60 , 32
360 DATA 126 , 35 , 94 , 35 , 86 , 35 , 78 , 35 , 70 , 35 , 229 , 205 , 178 , 42 , 205 ,
     227 , 45 , 225 , 17 , 58 , 255 , 1 , 15 , 0 , 205 , 60 , 32
380 DATA 62 , 0 , 30 , 0 , 86 , 35 , 78 , 35 , 6 , 0 , 229 , 205 , 178 , 42 , 205 , 227 ,
     45 , 225 , 17 , 73 , 255 , 1 , 15 , 0 , 205 , 60 , 32
395 DATA 126 , 198 , 48 , 215 , 35 , 62 , 13 , 215 , 195 , 102 , 253
400 DATA 126 , 238 , 192 , 215 , 35 , 94 , 35 , 86 , 213 , 35 , 70 , 197 , 62 , 40 , 215 ,
     35 , 197 , 62 , 0 , 30 , 0 , 86 , 35 , 78 , 35 , 6 , 0 , 229 , 205 , 178 , 42 , 205 , 227 ,
     45 , 225 , 193 , 120 , 254 , 1 , 40 , 3 , 62 , 44 , 215 , 16 , 226 , 62 , 41 , 215
450 DATA 193 , 209 , 27 , 27 , 27 , 16 , 252 , 62 , 13 , 215 , 122 , 179 , 27 , 27 , 27 ,
     27 , 27 , 202 , 102 , 253 , 213 , 126 , 35 , 94 , 35 , 86 , 35 , 78 , 35 , 70 , 35 , 229 ,
     205 , 178 , 42 , 205 , 227 , 45 , 225 , 209 , 24 , 221
500 DATA 126 , 238 , 224 , 215 , 35 , 126 , 203 , 127 , 40 , 249 , 203 , 191 , 215 ,
     195 , 0 , 255
600 DATA 126 , 215 , 62 , 36 , 215 , 62 , 61 , 215 , 62 , 34 , 215 , 35 , 78 , 35 , 70 ,
    35 , 120 , 177 , 11 , 40 , 5 , 126 , 35 , 215 , 24 , 246 , 62 , 34 , 215 , 62 , 13 , 215 ,
    195 , 102 , 253
700 DATA 126 , 203 , 175 , 215 , 62 , 61 , 215 , 35 , 126 , 35 , 94 , 35 , 86 , 35 , 78 ,
     35 , 70 , 35 , 229 , 205 , 178 , 42 , 205 , 227 , 45 , 225 , 62 , 13 , 215 , 195 , 102 ,
     253
 800 DATA 13 , 20 , 1, 86 , 97 , 108 , 111 , 114 , 20 , 0 , 13 , 20 , 1 , 76 , 105 , 109 ,
     105 , 116 , 101 , 20 , 0 , 13 , 20 , 1 , 80 , 97 , 115 , 111 , 20 , 0 , 13 , 20 , 1 , 78 ,
     46 , l00 , 101 , 32 , l08 , 105 , 110 , 101 , 97 , 20 , 0 , 13 , 20 , 1 , 78 , 46 , 100 ,
     101 , 32 , 111 , 114 , 100 , 101 , 110 , 20 , 0
1000 IF RAM=65535 THEN GO TO 1200
1010 POKE 32114,126
1020 POKE 32117,126
1030 POKE 32126,126
1040 POKE 32129,126
1050 POKE 32134,125
1060 POKE 32216,125
1070 POKE 32365,125
1080 POKE 32434,125
1090 POKE 32472,127
1100 POKE 32507,125
1110 POKE 32539,125
1120 REM POKES QUE FALTAN EN EL LIBRO Y SON NECESARIOS PARA 16K
1130 REM AJUSTE DE DIRECCIONES DE ETIQUETAS MENS (BYTE ALTO)
1140 POKE 32239,127
1150 POKE 32267,127
1160 POKE 32294,127
1170 POKE 32321,127
1180 POKE 32348,127
1200 PRINT "PULSA UNA TECLA PARA PONER EN MARCHA EL CODIGO
     MAQUINA": PAUSE 0: RANDOMIZE USR L1
1210 STOP
```

```
2000 LET RAM=PEEK 23732+256*PEEK 23733
2010 LET CL=64858
2020 LET L1=64859: LET L2=65367
2030 IF RAM<>32767 AND RAM<>65535 THEN STOP
2040 IF RAM=65535 THEN RETURN
2050 LET CL=32090
2060 LET L1=32091: LET L2=32599
2070 RETURN
```



APENDICES

Apéndice A

LECTURAS RECOMENDADAS

1. Mastering Machine Code on your Zx Spectrum
   por Toni Baker
   Interface Publications , U.K.

2. The Complete Spectrum ROM Disassembly
   por Dr.Ian Logan *y* Dr. Frank O'Hara
   Melbourne House Publishers ,U.K.

3. Programming the Z80
   por Rodney Zaks
   Sybex ,U.S.A.

4. Assembly Language for Arcade Games
   and other Fast Spectrum Programs
   por Stuart Nicholls
   Mc Graw-Hill Book Company ,U.K.

5. Revista mensual "Your Computer"
   Business Press International ,U.K.

TABLA DECIMAL - HEXADECIMAL - CARACTER

DEC	HEX	CARACTER
0	00	NO UTILIZADO
1	01	NO UTILIZADO
2	02	NO UTILIZADO
3	03	NO UTILIZADO
4	04	NO UTILIZADO
5	05	NO UTILIZADO
6	06	NO UTILIZADO
7	07	NO UTILIZADO
6	06	PRINT coma
7	07	EDIT
8	08	CURSOR izquierda
9	09	CURSOR derecha
10	0A	CURSOR abajo
11	0B	CURSOR arriba
12	0C	DELETE
13	0D	ENTER
14	0E	NUMERO
15	0F	NO UTILIZADO
16	10	CONTROL INK
17	11	CONTROL PAPER
18	12	CONTROL FLASH
19	13	CONTROL BRIGHT
20	14	CONTROL INVERSE
21	15	CONTROL OVER
22	16	CONTROL AT
23	17	CONTROL TAB
24	18	NO UTILIZADO
25	19	NO UTILIZADO
26	1A	NO UTILIZADO
27	1B	NO UTILIZADO
28	1C	NO UTILIZADO
29	1D	NO UTILIZADO
30	1E	NO UTILIZADO
31	1F	NO UTILIZADO
32	20	
33	21	'
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	´
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	–
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?
64	40	@
65	41	A
66	42	B

DEC	HEX	CARACTER
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	
95	5F	_
96	60	£
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	©
128	80	
129	81	▝
130	82	▘
131	83	▀
132	84	▗
133	85	▐
134	86	▚
135	87	▜

DEC	HEX	CARACTER
136	88	▖
137	89	▞
138	8A	▌
139	8B	▛
140	8C	▄
141	8D	▟
142	8E	▙
143	8F	█
144	90	A
145	91	B
146	92	C
147	93	D
148	94	E
149	95	F
150	96	G
151	97	H
152	98	I
153	99	J
154	9A	K
155	9B	L
156	9C	M
157	9D	N
158	9E	O
159	9F	P
160	A0	Q
161	A1	R
162	A2	S
163	A3	T
164	A4	U
165	A5	RND
166	A6	INKEY$
167	A7	PI
168	A8	FN
169	A9	POINT
170	AA	SCREEN$
171	AB	ATTR
172	AC	AT
173	AD	TAB
174	AE	VAL$
175	AF	CODE
176	B0	VAL
177	B1	LEN
178	B2	SIN
179	B3	COS
180	B4	TAN
181	B5	ASN
182	B6	ACS
183	B7	ATN
184	B8	LN
185	B9	EXP
186	BA	INT
187	BB	SQR
188	BC	SGN
189	BD	ABS
190	BE	PEEK
191	BF	IN
192	C0	USR
193	C1	STR$
194	C2	CHR$
195	C3	NOT
196	C4	BIN
197	C5	OR
198	C6	AND
199	C7	<=
200	C8	>=
201	C9	<>
202	CA	LINE
203	CB	THEN
204	CC	TO
205	CD	STEP
206	CE	DEF FN
207	CF	CAT
208	D0	FORMAT
209	D1	MOVE
210	D2	ERASE
211	D3	OPEN #
212	D4	CLOSE #
213	D5	MERGE
214	D6	VERIFY
215	D7	BEEP
216	D8	CIRCLE
217	D9	INK
218	DA	PAPER
219	DB	FLASH
220	DC	BRIGHT
221	DD	INVERSE
222	DE	OVER
223	DF	OUT
224	E0	LPRINT
225	E1	LL IST
226	E2	STOP
227	E3	READ
228	E4	DATA
229	E5	RESTORE
230	E6	NEW
231	E7	BORDER
232	E8	CONTINUE
233	E9	DIM
234	EA	REM
235	EB	FOR
236	EC	GO TO
237	ED	GO SUB
238	EE	INPUT
239	EF	LOAD
240	F0	LIST
241	F1	LET
242	F2	PAUSE
243	F3	NEXT
244	F4	POKE
245	F5	PRINT
246	F6	PLOT
247	F7	RUN
248	F8	SAVE
249	F9	RANDOMIZE
250	FA	I F
251	FB	CLS
252	FC	DRAW
253	FD	CLEAR
254	FE	RETURN
255	FF	COPY

TABLA DEC - HEX - COMPLEMENTO A DOS

DEC	HEX	COMPL A DOS	DEC	HEX	COMPL A DOS
0	00	0	42	2A	42
1	01	1	43	2B	43
2	02	2	44	2C	44
3	03	3	45	2D	45
4	04	4	46	2E	46
5	05	5	47	2F	47
6	06	6	48	30	48
7	07	7	49	31	49
8	08	8	50	32	50
9	09	9	51	33	51
10	0A	10	52	34	52
11	0B	11	53	35	53
12	0C	12	54	36	54
13	0D	13	55	37	55
14	0E	14	56	38	56
15	0F	15	57	39	57
16	10	16	58	3A	58
17	11	17	59	3B	59
18	12	18	60	3C	60
19	13	19	61	3D	61
20	14	20	62	3E	62
21	15	21	63	3F	63
22	16	22	64	40	64
23	17	23	65	41	65
24	18	24	66	42	66
25	19	25	67	43	67
26	1A	26	68	44	68
27	1B	27	69	45	69
28	1C	28	70	46	70
29	1D	29	71	47	71
30	1E	30	72	48	72
31	1F	31	73	49	73
32	20	32	74	4A	74
33	21	33	75	4B	75
34	22	34	76	4C	76
35	23	35	77	4D	77
36	24	36	78	4E	78
37	25	37	79	4F	79
38	26	38	80	50	80
39	27	39	81	51	81
40	28	40	82	52	82
41	29	41	83	53	83

DEC	HEX	COMPL A DOS	DEC	HEX	COMPL A DOS
84	54	84	133	85	-123
85	55	85	134	86	-122
86	56	86	135	87	-121
87	57	87	136	88	-120
88	58	88	137	89	-119
89	59	89	138	8A	-118
90	5A	90	139	8B	-117
91	5B	91	140	8C	-116
92	5C	92	141	8D	-115
93	5D	93	142	8E	-114
94	5E	94	143	8F	-113
95	5F	95	144	90	-112
96	60	96	145	91	-111
97	61	97	146	92	-110
98	62	98	147	93	-109
99	63	99	148	94	-108
100	64	100	149	95	-107
101	65	101	150	96	-106
102	66	102	151	97	-105
103	67	103	152	98	-104
104	68	104	153	99	-103
105	69	105	154	9A	-102
106	6A	106	155	9B	-101
107	6B	107	156	9C	-100
108	6C	108	157	9D	-99
109	6D	109	158	9E	-98
110	6E	110	159	9F	-97
111	6F	111	160	A0	-96
112	70	112	161	Al	-95
113	71	113	162	A2	-94
114	72	114	163	A3	-93
115	73	115	164	A4	-92
116	74	116	165	A5	-91
117	75	117	166	A6	-90
118	76	118	167	A7	-89
119	77	119	168	A8	-88
120	78	120	169	A9	-87
121	79	121	170	AA	-86
122	7A	122	171	AB	-85
123	7B	123	172	AC	-84
124	7C	124	173	AD	-83
125	7D	125	174	AE	-82
126	7E	126	175	AF	-81
127	7F	127	176	B0	-80
128	80	-128	177	B1	-79
129	81	-127	178	B2	-78
130	82	-126	179	B3	-77
131	83	-125	180	B4	-76
132	84	-124	181	B5	-75

DEC	HEX	COMPL A DOS
182	B6	-74
183	B7	-73
184	B8	-72
185	B9	-71
186	BA	-70
187	BB	-69
188	BC	-68
189	BD	-67
190	BE	-66
191	BF	-65
192	C0	-64
193	Cl	-63
194	C2	-62
195	C3	-61
196	C4	-60
197	C5	-59
198	C6	-58
199	C7	-57
200	C8	-56
201	C9	-55
202	CA	-54
203	CB	-53
204	CC	-52
205	CD	-51
206	CE	-50
207	CF	-49
208	D0	-48
209	D1	-47
210	D2	-46
211	D3	-45
212	D4	-44
213	D5	-43
214	D6	-42
215	D7	-41
216	D8	-40
217	D9	-39
218	DA	-38
219	DB	-37
220	DC	-36
221	DD	-35
222	DE	-34
223	DF	-33
224	E0	-32
225	El	-31
226	E2	-30
227	E3	-29
228	E4	-28
229	E5	-27
230	E6	-26
231	E7	-25
232	E8	-24
233	E9	-23
234	EA	-22
235	EB	-21
236	EC	-20
237	ED	-19
238	EE	-18
239	EF	-17
240	F0	-16
241	F1	-15
242	F2	-14
243	F3	-13
244	F4	-12
245	F5	-11
246	F6	-10
247	F7	-9
248	F8	-8
249	F9	-7
250	FA	-6
251	FB	-5
252	FC	-4
253	FD	-3
254	FE	-2
255	FF	-1

INSTRUCCIONES DE LA CPU Z80 CLASIFICADAS POR EL CODIGO OBJETO

N - Valor de 8 bits en el rango 0 a 255
NN - Valor de 16 bits en el rango 0 a 65535
DES - Valor de 8 bits en el rango -128 a +127

TABLA GENERAL

CODIGO OBJETO	MNEMONICO		CODIGO OBJETO	MNEMONICO
00	NOP	+	10 XX	DJNZ,DES
01 XX XX	LD BC, (NN)	+	11 XX XX	LD DE,NN
02	LD (BC),A	+	12	LD (DE),A
03	INC BC	+	13	INC DE
04	INC B	+	14	INC D
05	DEC B	+	15	DEC D
06 XX	LD B,N	+	16 XX	LD D,N
07	RLCA	+	17	RLA
08	EX AF,A'F'	+	18 XX	JR,DES
09	ADD HL,BC	+	19	ADD HL,DE
0A	LD A, (BC)	+	1A	LA A, (DE)
0B	DEC BC	+	1B	DEC DE
0C	INC C	+	1C	INC E
0D	DEC C	+	1D	DEC E
0E XX	LD C,N	+	1E XX	LD E,N
0F	RRCA	+	1F	RRA
20 XX	JR NZ,DES	+	30 XX	JR NC,DES
21 XX XX	LD HL,NN	+	31 XX XX	LD SP,NN
22 XX XX	LD (NN),HL	+	32 XX XX	LD (NN),A
23	INC HL	+	33	INC SP
24	INC H	+	34	INC (HL)
25	DEC H	+	35	DEC (HL)
26 XX	LD H,N	+	36 XX	LD (HL),N
27	DAA	+	37	SCF
28 XX	JR Z,DES	+	38 XX	JR C,DES
29	ADD HL,HL	+	39	ADD HL,SP
2A XX XX	LD HL, (NN)	+	3A XX XX	LD A, (NN)
2B	DEC HL	+	3B	DEC SP
2C	INC L	+	3C	INC A
2D	DEC L	+	3D	DEC A
2E XX	LD L,N	+	3E XX	LD A,N
2F	CPL	+	3F	CCF

40	LD B,B	+	50	LD D,B
41	LD B,C	+	51	LD D,C
42	LD B,D	+	52	LD D,D
43	LD B,E	+	53	LD D,E
44	LDB,H	+	54	LD D,H
45	LD B,L	+	55	LD D,L
46	LD B, (HL)	+	56	LD D, (HL)
47	LD B,A	+	57	LD D,A
48	LD C,B	+	58	LD E,B
49	LD C,C	+	59	LD E,C
4A	LD C,D	+	5A	LD E,D
4B	LD C,E	+	5B	LD E,E
4C	LD C,H	+	5C	LD E,H
4D	LD C,L	+	5D	LD E,L
4E	LD C, (HL)	+	5E	LD E, (HL)
4F	LD C,A	+	5F	LD E,A

60	LD H,B	+	70	LD (HL),B
61	LD H,C	+	71	LD (HL),C
62	LD H,D	+	72	LD (HL),D
63	LD H,E	+	73	LD (HL) ,E
64	LD H,H	+	74	LD (HL),H
65	LD H,L	+	75	LD (HL),L
66	LD H, (HL)	+	76	HALT
67	LD H,A	+	77	LD (HL) ,A
68	LD L,B	+	78	LD A,B
69	LD L,C	+	79	LD A,C
6A	LD L,D	+	7A	LD A,D
6B	LD L,E	+	7B	LD A,E
6C	LD L,H	+	7C	LD A,H
6D	LD L,L	+	7D	LD A,L
6E	LD L, (HL)	+	7E	LD A, (HL)
6F	LD L,A	+	7F	LD A,A

80	ADD A,B	+	90	SUB B
81	ADD A,C	+	91	SUB C
82	ADD A,D	+	92	SUB D
83	ADD A,E	+	93	SUB E
84	ADD A,H	+	94	SUB H
85	ADD A,L	+	95	SUB L
86	ADD A, (HL)	+	96	SUB (HL)
87	ADD A,A	+	97	SUB A
88	ADC A,B	+	98	SBC A,B
89	ADC A,C	+	99	SBC A,C
8A	ADC A,D	+	9A	SBC A,D
8B	ADC A,E	+	9B	SBC A,E
8C	ADC A,H	+	9C	SBC A,H
8D	ADC A,L	+	9D	SBC A,L
8E	ADC A, (HL)	+	9E	SBC A, (HL)
8F	ADC A,A	+	9F	SBC A,A

A0	AND B	+	B0	OR B
Al	AND C	+	B1	OR C
A2	AND D	+	B2	OR D
A3	AND E	+	B3	OR E
A4	AND H	+	B4	OR H
A5	AND L	+	B5	OR L
A6	AND (HL)	+	B6	OR (HL)
A7	AND A	+	B7	OR A
A8	XOR B	+	B8	CP B
A9	XOR C	+	B9	CP C
AA	XOR D	+	BA	CP D
AB	XOR E	+	BB	CP E
AC	XOR H	+	BC	CP H
AD	XOR L	+	BD	CP L
AE	XOR (HL)	+	BE	CP (HL)
AF	XOR A	+	BF	CP A

C0	RET NZ	+	D0	RET NC
C1	POP BC	+	D1	POP DE
C2 XX XX	JP NZ,NN	+	D2 XX XX	JP NC,NN
C3 XX XX	JP NN	+	D3 XX	OUT (N),A
C4 XX XX	CALL NZ,NN	+	D4 XX XX	CALL NC,NN
C5	PUSH BC	+	D5	PUSH DE
C6 XX	ADD A,N	+	D6 XX	SUB N
C7	RST 00H	+	D7	RST 10H
C8	RET Z	+	D8	RET C
C9	RET	+	D9	EXX
CA XX XX	JP Z,NN	+	DA XX XX	JP C,NN
CB	VER TABLA CB	+	DB XX	IN A, (N)
CC XX XX	CALL Z,NN	+	DC XX XX	CALL C,NN
CD XX XX	CALL NN	+	DD	VER TABLA DD
CE XX	ADC A,N	+	DE XX	SBC A,N
CF	RST 08H	+	DF	RST 18H

E0	RET PO	+	F0	RET P
El	POP HL	+	F1	POP AF
E2 XX XX	JP PO,NN	+	F2 XX XX	JP P,NN
E3	EX (SP),HL	+	F3	DI
E4 XX XX	CALL PO,NN	+	F4 XX XX	CALL P,NN
E5	PUSH HL	+	F5	PUSH AF
E6 XX	AND N	+	F6 XX	OR N
E7	RST 20H	+	F7	RST 30H
E8	RET PE	+	F8	RET M
E9	JP (HL)	+	F9	LD SP,HL
EA XX XX	JP PE,NN	+	FA XX XX	JP M,NN
EB	EX DE,HL	+	FB	EI
EC XX XX	CALL PE,NN	+	FC X X XX	CALL M,NN
ED	VER TABLA ED	+	FD	VER TABLA FD
EE XX	XOR N	+	FE XX	CP N
EF	RST 28H	+	FF	RST 38H

TABLA CB

CB 00	RLC B	+	CB 10	RL B
CB 01	RLC C	+	CB 11	RL C
CB 02	RLC D	+	CB 12	RL D
CB 03	RLC E	+	CB 13	RL E
CB 04	RLC H	+	CB 14	RL H
CB 05	RLC L	+	CB 15	RL L
CB 06	RLC (HL)	+	CB 16	RL (HL)
CB 07	RLC A	+	CB 17	RL A
CB 08	RRC B	+	CB 18	RR B
CB 09	RRC C	+	CB 19	RR C
CB 0A	RRC D	+	CB 1A	RR D
CB 0B	RRC E	+	CB 1B	RR E
CB 0C	RRC H	+	CB 1C	RR H
CB 0D	RRC L	+	CB 1D	RR L
CB 0E	RRC (HL)	+	CB 1E	RR (HL)
CB 0F	RRC A	+	CB 1F	RR A

CB 20	SLA B	+		
CB 21	SLA C	+		
CB 22	SLA D	+		
CB 23	SLA E	+		
CB 24	SLA H	+		
CB 25	SLA L	+		
CB 26	SLA (HL)	+		
CB 27	SLA A	+		
CB 28	SRA B	+	CB 38	SRL B
CB 29	SRA C	+	CB 39	SRL C
CB 2A	SRA D	+	CB 3A	SRL D
CB 2B	SRA E	+	CB 3B	SRL E
CB 2C	SRA H	+	CB 3C	SRL H
CB 2D	SRA L	+	CB 3D	SRL L
CB 2E	SRA (HL)	+	CB 3E	SRL (HL)
CB 2F	SRA A	+	CB 3F	SRL A

CB 40	BIT 0,B	+	CB 50	BIT 2,B
CB 41	BIT 0,C	+	CB 51	BIT 2,C
CB 42	BIT 0,D	+	CB 52	BIT 2,D
CB 43	BIT 0,E	+	CB 53	BIT 2,E
CB 44	BIT 0,H	+	CB 54	BIT 2,H
CB 45	BIT 0,L	+	CB 55	BIT 2,L
CB 46	BIT 0, (HL)	+	CB 56	BIT 2, (HL)
CB 47	BIT 0,A	+	CB 57	BIT 2,A
CB 48	BIT 1,B	+	CB 58	BIT 3,B
CB 49	BIT 1,C	+	CB 59	BIT 3,C
CB 4A	BIT 1,D	+	CB 5A	BIT 3,D
CB 4B	BIT 1,E	+	CB 5B	BIT 3,E
CB 4C	BIT 1,H	+	CB 5C	BIT 3,H
CB 4D	BIT 1,L	+	CB 5D	BIT 3,L
CB 4E	BIT 1, (HL)	+	CB 5E	BIT 3, (HL)
CB 4F	BIT 1,A	+	CB 5F	BIT 3,A

CB 60	BIT 4,B	+	CB 70	BIT 6,B
CB 61	BIT 4,C	+	CB 71	BIT 6,C
CB 62	BIT 4,D	+	CB 72	BIT 6,D
CB 63	BIT 4,E	+	CB 73	BIT 6,E
CB 64	BIT 4,H	+	CB 74	BIT 6,H
CB 65	BIT 4,L	+	CB 75	BIT 6,L
CB 66	BIT 4, (HL)	+	CB 76	BIT 6, (HL)
CB 67	BIT 4,A	+	CB 77	BIT 6,A
CB 68	BIT 5,B	+	CB 78	BIT 7,B
CB 69	BIT 5,C	+	CB 79	BIT 7,C
CB 6A	BIT 5,D	+	CB 7A	BIT 7,D
CB 6B	BIT 5,E	+	CB 7B	BIT 7,E
CB 6C	BIT 5,H	+	CB 7C	BIT 7,H
CB 6D	BIT 5,L	+	CB 7D	BIT 7,L
CB 6E	BIT 5, (HL)	+	CB 7E	BIT 7, (HL)
CB 6F	BIT 5,A	+	CB 7F	BIT 7,A

CB 80	RES 0,B	+	CB 90	RES 2,B
CB 81	RES 0,C	+	CB 91	RES 2,C
CB 82	RES 0,D	+	CB 92	RES 2,D
CB 83	RES 0,E	+	CB 93	RES 2,E
CB 84	RES 0,H	+	CB94	RES 2,H
CB 85	RES 0,L	+	CB 95	RES 2,L
CB 86	RES 0, (HL)	+	CB 96	RES 2, (HL)
CB 87	RES 0,A	+	GB 97	RES 2,A
CB 88	RES 1,B	+	CB 98	RES 3,B
CB 89	RES 1,C	+	CB 99	RES 3,C
CB 8A	RES 1,D	+	CB 9A	RES 3,D
CB 8B	RES 1,E	+	CB 9B	RES 3,E
CB 8C	RES 1,H	+	CB 9C	RES 3,H
CB 8D	RES 1,L	+	CB 9D	RES 3,L
CB 8E	RES 1, (HL)	+	CB 9E	RES 3, (HL)
CB 8F	RES 1,A	+	CB 9F	RES 3,A

CB A0	RES 4,B	+	CB B0	RES 6,B
CB Al	RES 4,C	+	CB B1	RES 6,C
CB A2	RES 4,D	+	CB B2	RES 6,D
CB A3	RES 4,E	+	CB B3	RES 6,E
CB A4	RES 4,H	+	CB B4	RES 6,H
CB A5	RES 4,L	+	CB B5	RES 6,L
CB A6	RES 4, (HL)	+	CB B6	RES 6, (HL)
CB A7	RES 4,A	+	CB B7	RES 6,A
CB A8	RES 5,B	+	CB B8	RES 7,B
CB A9	RES 5,C	+	CB B9	RES 7,C
CB AA	RES 5,D	+	CB BA	RES 7,D
CB AB	RES 5,E	+	CB BB	RES 7,E
CB AC	RES 5,H	+	CB BC	RES 7,H
CB AD	RES 5,L	+	CB BD	RES 7,L
CB AE	RES 5, (HL)	+	CB BE	RES 7, (HL)
CB AF	RES 5,A	+	CB BF	RES 7,A

CB C0	SET 0,B	+	CB D0	SET 2,B
CB C1	SET 0,C	+	CB D1	SET 2,C
CB C2	SET 0,D	+	CB D2	SET 2,D
CB C3	SET 0,E	+	CB D3	SET 2,E
CB C4	SET 0,H	+	CB D4	SET 2,H
CB C5	SET 0,L	+	CB D5	SET 2,L
CB C6	SET 0, (HL)	+	CB D6	SET 2, (HL)
CB C7	SET 0,A	+	CB D7	SET 2,A
CB C8	SET 1,B	+	CB D8	SET 3,B
CB C9	SET 1,C	+	CB D9	SET 3,C
CB CA	SET 1,D	+	CB DA	SET 3,D
CB CB	SET 1,E	+	CB DB	SET 3,E
CB CC	SET 1,H	+	CB DC	SET 3,H
CB CD	SET 1,L	+	CB DD	SET 3,L
CB CE	SET 1, (HL)	+	CB DE	SET 3, (HL)
CB CF	SET 1,A	+	CB DF	SET 3,A

CB E0	SET 4,B	+	CB F0	SET 6,B
CB El	SET 4,C	+	CB F1	SET 6,C
CB E2	SET 4,D	+	CB F2	SET 6,D
CB E3	SET 4,E	+	CB F3	SET 6,E
CB E4	SET 4,H	+	CB F4	SET 6,H
CB E5	SET 4,L	+	CB F5	SET 6,L
CB E6	SET 4, (HL)	+	CB F6	SET 6, (HL)
CB E7	SET 4,A	+	CB F7	SET 6,A
CB E8	SET 5,B	+	CB F8	SET 7,B
CB E9	SET 5,C	+	CB F9	SET 7,C
CB EA	SET 5,D	+	CB FA	SET 7,D
CB EB	SET 5,E	+	CB FB	SET 7,E
CB EC	SET 5,H	+	CB FC	SET 7,H
CB ED	SET 5,L	+	CB FD	SET 7,L
CB EE	SET 5, (HL)	+	CB FE	SET 7, (HL)
CB EF	SET 5,A	+	CB FF	SET 7,A

TABLA DD

DD 09	ADD IX,BC	+	DD 72 DES	LD (IX+DES),D
DD 19	ADD IX,DE	+	DD 73 DES	LD (IX+DES),E
DD 21 XX XX	LD IX,NN	+	DD 74 DES	LD (IX+DES),H
DD 22 XX XX	LD(NN),IX	+	DD 75 DES	LD (IX+DES),L
DD 23	INC IX	+	DD 77 DES	LD (IX+DES),A
DD 29	ADD IX,IX	+	DD 7E DES	LD A, (IX+DES)
DD 2A XX XX	LD IX, (NN)	+	DD 86 DES	ADD A, (IX+DES)
DD 2B	DEC IX	+	DD 8E DES	ADC A, (IX+DES)
DD 34 DES	INC (IX+DES)	+	DD 96 DES	SUB (IX+DES)
DD 35 DES	DEC (IX+DES)	+	DD 9E DES	SBC A, (IX+DES)
DD 36 DES XX	LD (IX+DES),N	+	DD E6 DES	AND (IX+DES)
DD 39	ADD IX,SP	+	DD AE DES	XOR (IX+DES)
DD 46 DES	LD B, (IX+DES)	+	DD B6 DES	OR (IX+DES)
DD 4E DES	LD C, (IX+DES)	+	DD BE DES	CP (IX+DES)
DD 56 DES	LD D, (IX+DES)	+	DD El	POP IX
DD 5E DES	LD E, (IX+DES)	+	DD E3	EX (SP),IX
DD 66 DES	LD H, (IX+DES)	+	DD E5	PUSH IX
DD 6E DES	LD L, (IX+DES)	+	DD E9	JP (IX)
DD 70 DES	LD (IX+DES),B	+	DD F9	LD SP,IX
DD 71 DES	LD (IX+DES),C	+		

DD CB DES 06	RLC (IX+DES)	+	DD CB DES 8E	RES 1, (IX+DES)
DD CB DES 0E	RRC (IX+DES)	+	DD CB DES 96	RES 2, (IX+DES)
DD CB DES 16	RL (IX+DES)	+	DD CB DES 9E	RES 3, (IX+DES)
DD CB DES 1E	RR (IX+DES)	+	DD CB DES A6	RES 4, (IX+DES)
DD CB DES 26	SLA (IX+DES)	+	DD CB DES AE	RES 5, (IX+DES)
DD CB DES 2E	SRA (IX+DES)	+	DD CB DES B6	RES 6, (IX+DES)
DD CB DES 3E	SRL (IX+DES)	+	DD CB DES BE	RES 7, (IX+DES)
DD CB DES 46	BIT 0, (IX+DES)	+	DD CB DES C6	SET 0, (IX+DES)
DD CB DES 4E	BIT 1, (IX+DES)	+	DD CB DES CE	SET 1, (IX+DES)
DD CB DES 56	BIT 2, (IX+DES)	+	DD CB DES D6	SET 2, (IX+DES)
DD CB DES 5E	BIT 3, (IX+DES)	+	DD CB DES DE	SET 3, (IX+DES)
DD CB DES 66	BIT 4, (IX+DES)	+	DD CB DES E6	SET 4, (IX+DES)
DD CB DES 6E	BIT 5, (IX+DES)	+	DD CB DES EE	SET 5, (IX+DES)
DD CB DES 76	BIT 6, (IX+DES)	+	DD CB DES F6	SET 6, (IX+DES)
DD CB DES 7E	BIT 7, (IX+DES)	+	DD CB DES FE	SET 7, (IX+DES)
DD CB DES 86	RES 0, (IX+DES)	+		

TABLA ED

ED 40	IN B, (C)	+	ED 50	IN D, (C)
ED 41	OUT (C),B	+	ED 51	OUT (C),D
ED 42	SBC HL,BC	+	ED 52	SBC HL,DE
ED 43 XX XX	LD (NN),BC	+	ED 53 XX XX	LD (NN),DE
ED 44	NEG	+		
ED 45	RETN	+		
ED 46	IM 0	+	ED 56	IM 1
ED 47	LD I,A	+	ED 57	LD A,I
ED 48	IN C, (C)	+	ED 58	IN E, (C)
ED 49	OUT (C),C	+	ED 59	OUT (C) ,E
ED 4A	ADC HL,BC	+	ED 5A	ADC HL,DE
ED 4B XX XX	LD BC, (NN)	+	ED 5B XX XX	LD DE, (NN)
ED 4D	RETI	+	ED 5E	IM 2
ED 4F	LD R,A	+	ED 5F	LD A,R

ED 60	IN H, (C)	+	ED A0	LDI
ED 61	OUT (C),H	+	ED Al	CPI
ED 62	SBC HL,HL	+	ED A2	INI
ED 63 XX XX	LD (NN),HL	+	ED A3	OUTI
ED 67	RRD	+	ED A8	LDD
ED 68	IN L, (C)	+	ED A9	CPD
ED 69	OUT (C),L	+	ED AA	IND
ED 6A	ADC HL,HL	+	ED AB	OUTD
ED 6B	LD HL, (NN)	+	ED B0	LDIR
ED 6F	RLD	+	ED B1	CPIR
ED 72	SBC HL,SP	+	ED B2	INIR
ED 73 XX XX	LD (NN),SP	+	ED B3	OTIR
ED 78	IN A, (C)	+	ED B8	LDDR
ED 79	OUT (C),A	+	ED B9	CPDR
ED 7A	ADC HL,SP	+	ED BA	INDR
ED 7B XX XX	LD SP, (NN)	+	ED BB	OTRD

<u>TABLA FD</u>

FD 09	ADD IY,BC	+
FD 19	ADD IY,DE	+
FD 21 XX XX	LD IY,NN	+
FD 22 XX XX	LD (NN),IY	+
FD 23	INC IY	+
FD 29	ADD IY,IY	+
FD 2A XX XX	LD IY, (NN)	+
FD 2B	DEC IY	+
FD 34 DES	INC (IY+DES)	+
FD 35 DES	DEC (IY+DES)	+
FD 36 DES XX	LD (IY+DES),N	+
FD 39	ADD IY,SP	+
FD 46 DES	LD B, (IY+DES)	+
FD 4E DES	LD C, (IY+DES)	+
FD 56 DES	LD D, (IY+DES)	+
FD 5E DES	LD E, (IY+DES)	+
FD 66 DES	LD H, (IY+DES)	+
FD 6E DES	LD L, (IY+DES)	+
FD 70 DES	LD (IY+DES),B	+
FD 71 DES	LD (IY+DES),C	+
FD 72 DES	LD (IY+DES),D	
FD 73 DES	LD (IY+DES),E	
FD 74 DES	LD (IY+DES),H	
FD 75 DES	LD (IY+DES),L	
FD 77 DES	LD (IY+DES),A	
FD 7E DES	LD A, (IY+DES)	
FD 86 DES	ADD A, (IY+DES)	
FD 8E DES	ADC A, (IY+DES)	
FD 96 DES	SUB (IY+DES)	
FD 9E DES	SBC A, (IY+DES)	
FD E6 DES	AND (IY+DES)	
FD AE DES	XOR (IY+DES)	
FD B6 DES	OR (IY+DES)	
FD BE DES	CP (IY+DES)	
FD El	POP IY	
FD E3	EX (SP),IY	
FD E5	PUSH IY	
FD E9	JP (IY)	
FD F9	LD SP,IY	

FD CB DES 06	RLC (IY+DES)	+
FD CB DES 0E	RRC (IY+DES)	+
FD CB DES 16	RL (IY+DES)	+
FD CB DES 1E	RR (IY+DES)	+
FD CB DES 26	SLA (IY+DES)	+
FD CB DES 2E	SRA (IY+DES)	+
FD CB DES 3E	SRL (IY+DES)	+
FD CB DES 46	BIT 0, (IY+DES)	+
FD CB DES 4E	BIT 1, (IY+DES)	+
FD CB DES 56	BIT 2, (IY+DES)	+
FD CB DES 5E	BIT 3, (IY+DES)	+
FD CB DES 66	BIT 4, (IY+DES)	+
FD CB DES 6E	BIT 5, (IY+DES)	+
FD CB DES 76	BIT 6, (IY+DES)	+
FD CB DES 7E	BIT 7, (IY+DES)	+
FD CB DES 86	RES 0, (IY+DES)	+
FD CB DES 8E	RES 1, (IY+DES)	
FD CB DES 96	RES 2, (IY+DES)	
FD CB DES 9E	RES 3, (IY+DES)	
FD CB DES A6	RES 4, (IY+DES)	
FD CB DES AE	RES 5, (IY+DES)	
FD CB DES B6	RES 6, (IY+DES)	
FD CB DES BE	RES 7, (IY+DES)	
FD CB DES C6	SET 0, (IY+DES)	
FD CB DES CE	SET 1, (IY+DES)	
FD CB DES D6	SET 2, (IY+DES)	
FD CB DES DE	SET 3, (IY+DES)	
FD CB DES E6	SET 4, (IY+DES)	
FD CB DES EE	SET 5, (IY+DES)	
FD CB DES F6	SET 6, (IY+DES)	
FD CB DES FE	SET 7, (IY+DES)	

INSTRUCCIONES DE LA CPU Z80 CLASIFICADAS POR MNEMONICO

N - Valor de 8 bits en el rango 0 a 255
NN - Valor de 16 bits en el rango 0 a 65535
DES - Valor de 8 bits en el rango -128 a +127

MNEMONICO	CODIGO OBJETO		MNEMONICO	CODIGO OBJETO
ADC A,HL	8E	+	AND (HL)	A6
ADC A, (IX+DES)	DD 8E DES	+	AND (IX+DES)	DD A6 DES
ADC A, (IY+DES)	FD 8E DES	+	AND (IY+DES)	FD A6 DES
ADC A,A	8F	+	AND A	A7
ADC A,B	88	+	AND B	A0
ADC A,C	89	+	AND C	Al
ADC A,D	8A	+	AND D	A2
ADC A,E	8B	+	AND E	A3
ADC A,H	8C	+	AND H	A4
ADC A,L	8D	+	AND L	A5
ADC A,N	CE XX	+	AND N	E6 XX
ADC HL,BC	ED 4A	+	BIT 0, (HL)	CB 46
ADC HL,DE	ED 5A	+	BIT 0, (IX+DES)	DD CB DES 46
ADC HL,HL	ED 6A	+	BIT 0, (IY+DES)	FD CB DES 46
ADC HL,SP	ED 7A	+	BIT 0,A	CB 47
ADD A, (HL)	86	+	BIT 0,B	CB 40
ADD A, (IX+DES)	DD 86 DES	+	BIT 0,C	CB 41
ADD A, (IY+DES)	FD 86 DES	+	BIT 0,D	CB 42
ADD A,A	87	+	BIT 0,E	CB 43
ADD A,B	80	+	BIT 0,H	CB 44
ADD A,C	81	+	BIT 0,L	CB 45
ADD A,D	82	+	BIT 1, (HL)	CB 4E
ADD A,E	83	+	BIT 1, (IX+DES)	DD CB DES 4E
ADD A,H	84	+	BIT 1, (IX+DES)	FD CB DES 4E
ADD A,L	85	+	BIT 1,A	CB 4F
ADD A,N	C6 XX	+	BIT 1,B	CB 48
ADD HL,BC	09	+	BIT 1,C	CB 49
ADD HL,DE	19	+	BIT 1,D	CB 4A
ADD HL,HL	29	+	BIT 1,E	CB 4B
ADD HL,SP	39	+	BIT 1,H	CB 4C
ADD IX,BC	DD 09	+	BIT 1,L	CB 4D
ADD IX,DE	DD 19	+	BIT 2, (HL)	CB 56
ADD IX,IX	DD 29	+	BIT 2, (IX+DES)	DD CB DES 56
ADD IX,SP	DD 39	+	BIT 2, (IY+DES)	FD CB DES 56
ADD IY,BC	FD 09	+	BIT 2,A	CB 57
ADD IY,DE	FD 19	+	BIT 2,B	CB 50
ADD IY,IY	FD 29	+	BIT 2,C	CB 51
ADD IY,SP	FD 39	+	BIT 2,D	CB 52

BIT 2,E	CB 53	+	CALL C,NN	CD XX XX
BIT 2,H	CB 54	+	CALL M,NN	DC XX XX
BIT 2,L	CB 55	+	CALL NC,NN	D4 XX XX
BIT 3, (HL)	CB 5E	+	CALL NN	CD XX XX
BIT 3, (IX+DES)	DD CB DES 5E	+	CALL NZ,NN	C4 XX XX
BIT 3, (IY+DES)	FD CB DES 5E	+	CALL P,NN	F4 XX XX
BIT 3,A	CB 5F	+	CALL PE,NN	EC XX XX
BIT 3,B	CB 58	+	CALL PO,NN	E4 XX XX
BIT 3,C	CB 59	+	CALL Z,NN	CC XX XX
BIT 3,D	CB 5A	+	CCF	3F
BIT 3,E	CB 5B	+	CP (HL)	BE
BIT 3,H	CB 5C	+	CP (IX+DES)	DD BE DES
BIT 3,L	CB 5D	+	CP (IY+DES)	FD BE DES
BIT 4, (HL)	CB 66	+	CP A	BF
BIT 4, (IX+DES)	DD CB DES 66	+	CP B	B8
BIT 4, (IY+DES)	FD CB DES 66	+	CP C	B9
BIT 4,A	CB 67	+	CP D	BA
BIT 4,B	CB 60	+	CP E	BB
BIT 4,C	CB 61	+	CP H	BC
BIT 4,D	CB 62	+	CP L	BD
BIT 4,E	CB 63	+	CP N	FE XX
BIT 4,H	CB 64	+	CPD	ED A9
BIT 4,L	CB 65	+	CPDR	ED B9
BIT 5, (HL)	CB 6E	+	CPI	ED Al
BIT 5, (IX+DES)	DD CB DES 6E	+	CPIR	ED B1
BIT 5, (IY+DES)	FD CB DES 6E	+	CPL	2F
BIT 5,A	CB 6F	+	DAA	27
BIT 5,B	CB 68	+	DEC (HL)	35
BIT 5,C	CB 69	+	DEC (IX+DES)	DD 35 DES
BIT 5,D	CB 6A	+	DEC. (IY+DES)	FD 35 DES
BIT 5,E	CB 6B	+	DEC A	3D
BIT 5,H	CB 6C	+	DEC B	05
BIT 5,L	CB 6D	+	DEC BC	0B
BIT 6, (HL)	CB 76	+	DEC C	0D
BIT 6, (IX+DES)	DD CB DES 76	+	DEC D	15
BIT 6,(IY+DES)	FD CB DES 76	+	DEC DE	1B
BIT 6,A	CB 77	+	DEC E	1D
BIT 6,B	CB 70	+	DEC H	25
BIT 6,C	CB 71	+	DEC HL	2B
BIT 6,D	CB 72	+	DEC IX	DD 2B
BIT 6,E	CB 73	+	DEC IY	FD 2B
BIT 6,H	CB 74	+	DEC L	2D
BIT 6,L	CB 75	+	DEC SP	3B
BIT 7, (HL)	CB 7E	+	DI	F3
BIT 7, (IX+DES)	DD CB DES 7E	+	DJ NZ,DES	10 DES
BIT 7, (IY+DES)	FD CB DES 7E	+	El	FB
BIT 7,A	CB 7F	+	EX (SP),HL	E3
BIT 7,B	CB 78	+	EX (SP),IX	DD E3
BIT 7,C	CB 79	+	EX (SP),IY	FD E3
BIT 7,D	CB 7A	+	EX AF,A'F'	08
BIT 7,E	CB 7B	+	EX DE,HL	EB
BIT 7,H	CB 7C	+	EXX	D9
BIT 7,L	CB 7D	+	HALT	76

IM 0	ED 46	+	LD (HL),B	70
IM 1	ED 56	+	LD (HL),C	71
IM 2	ED 5E	+	LD (HL),D	72
IN A, (C)	ED 78	+	LD (HL),E	73
IN A, (N)	DB XX	+	LD (HL),H	74
IN B, (C)	ED 40	+	LD (HL),L	75
IN C, (C)	ED 48	+	LD (HL),N	36 XX
IN D, (C)	ED 50	+	LD (IX+DES),A	DD 77 DES
IN E, (C)	ED 58	+	LD (IX+DES),B	DD 70 DES
IN H, (C)	ED 60	+	LD (IX+DES),C	DD 71 DES
IN L, (C)	ED 68	+	LD (IX+DES),D	DD 72 DES
INC (HL)	34	+	LD (IX+DES),E	DD 73 DES
INC (IX+DES)	DD 34 DES	+	LD (IX+DES),H	DD 74 DES
INC (IY+DES)	FD 34 DES	+	LD (IX+DES),L	DD 75 DES
INC A	3C	+	LD (IX+DES),N	DD 36 DES XX
INC B	04	+	LD (IY+DES),A	FD 77 DES
INC BC	03	+	LD (IY+DES),B	FD 70 DES
INC C	0C	+	LD (IY+DES),C	FD 71 DES
INC D	14	+	LD (IY+DES),D	FD 72 DES
INC DE	13	+	LD (IY+DES),E	FD 73 DES
INC E	1C	+	LD (IY+DES),H	FD 74 DES
INC H	24	+	LD (IY+DES),L	FD 75 DES
INC HL	23	+	LD (IY+DES),N	FD 36 DES XX
INC IX	DD 23	+	LD (NN),A	32 XX XX
INC IY	FD 23	+	LD (NN),BC	ED 43 XX XX
INC L	2C	+	LD (NN),DE	ED 53 XX XX
INC SP	33	+	LD (NN),HL	22 XX XX
IND	ED AA	+	LD (NN),IX	DD 22 XX XX
INDR	ED 8A	+	LD (NN),IY	FD 22 XX XX
INI	ED A2	+	LD (NN),SP	ED 73 XX XX
INIR	ED B2	+	LD A, (BC)	0A
JP (HL)	E9	+	LD A, (DE)	1A
JP (IX)	DD E9	+	LD A, (HL)	7E
JP (IY)	FD E9	+	LD A, (IX+DES)	DD 7E DES
JP C,NN	DA XX XX	+	LD A, (IY+DES)	FD 7E DES
JP M,NN	FA XX XX	+	LD A, (NN)	3A XX XX
JP NC,NN	D2 XX XX	+	LD A,A	7F
JP NN	C3 XX XX	+	LD A,B	78
JP NZ,NN	C2 XX XX	+	LD A,C	79
JP P,NN	F2 XX XX	+	LD A,D	7A
JP PE,NN	EA XX XX	+	LD A,E	7B
JP PO,NN	E2 XX XX	+	LD A,H	7C
JP Z,NN	CA XX XX	+	LD A,I	ED 57
JR C,DES	38 DES	+	LD A,L	7D
JR DES	18 DES	+	LD A,N	3E XX
JR NC,DES	30 DES	+	LD A,R	ED 5F
JR NZ,DES	20 DES	+	LD B, (HL)	46
JR Z,DES	28 DES	+	LD B, (IX+DES)	DD 46 DES
LD (BC),A	02	+	LD B, (IY+DES)	FD 46 DES
LD (DE),A	12	+	LD B,A	47
LD (HL) ,A	77	+	LD B,B	40

LD B,C	41	+	LD H,H	64
LD B,D	42	+	LD H,L	65
LD B,E	43	+	LD H,N	26 XX
LD B,H	44	+	LD HL, (NN)	2A XX XX
LD B,L	45	+	LD HL,NN	21 XX XX
LD B,N	06 XX	+	LD I,A	ED 47
LD BC, (NN)	ED 4B XX XX	+	LD IX, (NN)	DD 2A XX XX
LD BC,NN	01 XX XX	+	LD IX,NN	DD 21 XX XX
LD C, (HL)	4E	+	LD IY, (NN)	FD 2A XX XX
LD C, (IX+DES)	DD 4E DES	+	LD IY,NN	FD 21 XX XX
LD C, (IY+DES)	FD 4E DES	+	LD L, (HL)	6E
LD C,A	4F	+	LD L, (IX+DES)	DD 6E XX
LD C,B	48	+	LD L, (IY+DES)	FD 6E XX
LD C,C	49	+	LD L,A	6F
LD C,D	4A	+	LD L,B	68
LD C,E	4B	+	LD L,C	69
LD C,H	4C	+	LD L,D	6A
LD C,L	4D	+	LD L,E	6B
LD C,N	3E XX	+	LD L,H	6C
LD D, (HL)	56	+	LD L,L	6D
LD D, (IX+DES)	DD 56 DES	+	LD L,N	2E XX
LD D, (IY+DES)	FD 56 DES	+	LD R,A	ED 4F
LD D,A	57	+	LD SP, (NN)	ED 7B XX XX
LD D,B	50	+	LD SP,HL	F9
LD D,C	51	+	LD SP,IX	DD F9
LD D,D	52	+	LD SP,IY	FD F9
LD D,E	53	+	LD SP,NN	31 XX XX
LD D,H	54	+	LDD	ED A8
LD D,L	55	+	LDDR	ED B8
LD D,N	16 XX	+	LDI	ED A3
LD DE, (NN)	ED 5B XX XX	+	LDIR	ED B0
LD DE,NN	11 XX XX	+	NEG	ED 44
LD E, (HL)	5E	+	NOP	00
UD E, (IX+DES)	DD 5E DES	+	OR (HL)	B6
LD E, (IY+DES)	FD 5E DES	+	OR (IX+DES)	DD B6 DES
LD E,A	5F	+	OR (IY+DES)	FD B6 DES
LD E,B	58	+	OR A	B7
LD E,C	59	+	OR B	B0
LD E,D	5A	+	OR C	B1
LD E,E	5B	+	OR D	B2
LD E,H	5C	+	OR E	B3
LD E,L	5D	+	OR H	B4
LD E,N	1E XX	+	OR L	B5
LD H, (HL)	66	+	OR N	F6 XX
LD H, (IX+DES)	DD 66 XX	+	OTDR	ED BB
LD H, (IY+DES)	FD 66 XX	+	OTIR	ED B3
LD H,A	67	+	OUT (C),A	ED 79
LD H,B	60	+	OUT (C),B	ED 41
LD H,C	61	+	OUT (C),C	ED 49
LD H,D	62	+	OUT (C),D	ED 51
LD H,E	63	+	OUT (C),E	ED 59

OUT (C),H	ED 61	+	RES 3,B	CB 98
OUT (C),L	ED 69	+	RES 3,C	CB 99
OUT (N),A	D3 XX	+	RES 3,D	CB 9A
OUTD	ED AB	+	RES 3,E	CB 9B
OUTI	ED A3	+	RES 3,H	CB 9C
POP AF	F1	+	RES 3,L	CB 9D
POP BC	C1	+	RES 4, (HL)	CB A6
POP DE	D1	+	RES 4, (IX+DES)	DD CB DES A6
POP HL	E1	+	RES 4, (IX+DES)	FD CB DES A6
POP IX	DD El	+	RES 4,A	CB A7
POP IY	FD El	+	RES 4,B	CB A0
PUSH AF	F5	+	RES 4,C	CB Al
PUSH BC	C5	+	RES 4,D	CB A2
PUSH DE	D5	+	RES 4,E	CB A3
PUSH HL	E5	+	RES 4,H	CB A4
PUSH IX	DD E5	+	RES 4,L	CB A5
PUSH IY	FD E5	+	RES 5, (HL)	CB AE
RES 0, (HL)	CB 86	+	RES 5, (IX+DES)	DD CB DES AE
RES 0, (IX+DES)	DD CB DES 86	+	RES 5, (IX+DES)	FD CB DES AE
RES 0, (IY+DES)	FD CB DES 86	+	RES 5,A	CB AF
RES 0,A	CB 87	+	RES 5,B	CB A8
RES 0,B	CB 80	+	RES 5,C	CB A9
RES 0,C	CB 81	+	RES 5,D	CB AA
RES 0,D	CB 82	+	RES 5,E	CB AB
RES 0,E	CB 83	+	RES 5,H	CB AC
RES 0,H	CB 84	+	RES 5,L	CB AD
RES 0,L	CB 85	+	RES 6, (HL)	CB B6
RES 1, (HL)	CB 8E	+	RES 6, (IX+DES)	DD CB DES B6
RES 1, (IX+DES)	DD CB DES 8E	+	RES 6, (IY+DES)	FD CB DES B6
RES 1, (IY+DES)	FD CB DES 8E	+	RES 6,A	CB B7
RES 1,A	CB 8F	+	RES 6,B	CB B0
RES 1,B	CB 88	+	RES 6,C	CB B1
RES 1,C	CB 89	+	RES 6,D	CB B2
RES 1,D	CB 8A	+	RES 6,E	CB B3
RES 1,E	CB 8B	+	RES 6,H	CB B4
RES 1,H	CB 8C	+	RES 6,L	CB B5
RES 1,L	CB 8D	+	RES 7, (HL)	CB BE
RES 2, (HL)	CB 96	+	RES 7, (IX+DES)	DD CB DES BE
RES 2, (IX+DES)	DD CB DES 96	+	RES 7, (IY+DES)	FD CB DES BE
RES 2, (IY+DES)	FD CB DES 96	+	RES 7,A	CB BF
RES 2,A	CB 97	+	RES 7,B	CB B8
RES 2,B	CB 90	+	RES 7,C	CB B9
RES 2,C	CB 91	+	RES 7,D	CB BA
RES 2,D	CB 92	+	RES 7,E	CB BB
RES 2,E	CB 93	+	RES 7,H	CB BC
RES 2,H	CB 94	+	RES 7,L	CB BD
RES 2,L	CB 95	+	RET	C9
RES 3, (HL)	CB 9E	+	RET C	D8
RES 3, (IX+DES)	DD CB DES 9E	+	RET M	F8
RES 3, (IY+DES)	FD CB DES 9E	+	RET NC	D0
RES 3,A	CB 9F	+	RET NZ	C0

RETI	ED 4D	+	RST 20H	E7
RETN	ED 45	+	RST 28H	EF
RL (HL)	CB 16	+	RST 30H	F7
RL (IX+DES)	DD CB DES 16	+	RST 38H	FF
RL (IY+DES)	FD CB DES 16	+	SBC A, (HL)	9E
RL A	CB 17	+	SBC A, (IX+DES)	DD CB DES 9E
RL B	CB 10	+	SBC A, (IY+DES)	FD CB DES 9E
RL C	CB 11	+	SBC A,A	9F
RL D	CB 12	+	SBC A,B	98
RL E	CB 13	+	SBC A,C	99
RL H	CB 14	+	SBC A,D	9A
RL L	CB 15	+	SBC A,E	9B
RLA	17	+	SBC A,H	9C
RLC (HL)	CB 06	+	SBC A,L	9D
RLC (IX+DES)	DD CB DES 06	+	SBC A,N	DE XX
RLC (IY+DES)	FD CB DES 06	+	SBC HL,BC	ED 42
RLC A	CB 07	+	SBC HL,DE	ED 52
RLC B	CB 00	+	SBC HL,HL	ED 62
RLC C	CB 01	+	SBC HL,SP	ED 72
RLC D	CB 02	+	SCF	37
RLC E	CB 03	+	SET 0, (HL)	CB C6
RLC H	CB 04	+	SET 0, (IX+DES)	DD CB DES C6
RLC L	CB 05	+	SET 0, (IY+DES)	FD CB DES C6
RLCA	07	+	SET 0,A	CB C7
RLD	ED 6F	+	SET 0,B	CB C0
RR (HL)	CB 1E	+	SET 0,C	CB C1
RR (IX+DES)	DD CB DES 1E	+	SET 0,D	CB C2
RR (IY+DES)	FD CB DES 1E	+	SET 0,E	CB C3
RR A	CB 1F	+	SET 0,H	CB C4
RR B	CB 18	+	SET 0,L	CB C5
RR C	CB 19	+	SET 1, (HL)	CB CE
RR D	CB 1A	+	SET 1, (IX+DES)	DD CB DES CE
RR E	CB 1B	+	SET 1, (IY+DES)	DD CB DES CE
RR H	CB 1C	+	SET 1,A	CB CF
RR L	CB lD	+	SET 1,B	CB C8
RRA	lF	+	SET 1,C	CB C9
RRC (HL)	CB 0E	+	SET 1,D	CB CA
RRC (IX+DES)	DD CB DES 0E	+	SET 1,E	CB CB
RRC (IY+DES)	FD CB DES 0E	+	SET 1,H	CB CC
RRC A	CB 0F	+	SET 1,L	CB CD
RRC B	CB 08	+	SET 2, (HL)	CB D6
RRC C	CB 09	+	SET 2, (IX+DES)	DD CB DES D6
RRC D	CB 0A	+	SET 2, (IY+DES)	FD CB DES D6
RRC E	CB 0B	+	SET 2,A	CB D7
RRC H	CB 0C	+	SET 2,B	CB D0
RRC L	CB 0D	+	SET 2,C	CB D1
RRCA	0F	+	SET 2,D	CB D2
RRD	ED 67	+	SET 2,E	CB D3
RST 00H	C7	+	SET 2,H	CB D4
RST 08H	CF	+	SET 2,L	CB D5
RST 10H	D7	+	SET 3, (HL)	CB DE
RST 18H	DF	+	SET 3, (IX+DES)	DD CB DES DE

SET 3, (IY+DES)	FD CB DES DE	+
SET 3,A	CB DF	+
SET 3,B	CB D8	+
SET 3,C	CB D9	+
SET 3,D	CB DA	+
SET 3,E	CB DB	+
SET 3,H	CB DC	+
SET 3,L	CB DD	+
SET 4, (HL)	CB E6	+
SET 4, (IX+DES)	DD CB DES E6	+
SET 4, (IY+DES)	FD CB DES E6	+
SET 4,A	CB E7	+
SET 4,B	CB E0	+
SET 4,C	CB El	+
SET 4,D	CB E2	+
SET 4,E	CB E3	+
SET 4,H	CB E4	+
SET 4,L	CB E5	+
SET 5, (HL)	CB EE	+
SET 5, (IX+DES)	DD CB DES EE	+
SET 5, (IY+DES)	FD CB DES EE	+
SET 5,A	CB EF	+
SET 5,B	CB E8	+
SET 5,C	CB E9	+
SET 5,D	CB EA	+
SET 5,E	CB EB	+
SET 5,H	CB EC	+
SET 5,L	CB ED	+
SET 6, (HL)	CB F6	+
SET 6, (IX+DES)	DD CB DES F6	+
SET 6, (IY+DES)	FD CB DES F6	+
SET 6,A	CB F7	+
SET 6,B	CB F8	+
SET 6,C	CB F9	+
SET 6,D	CB FA	+
SET 6,E	CB FB	+
SET 6,H	CB FC	+
SET 6,L	CB FD	+
SET 7, (HL)	CB FE	+
SET 7, (IX+DES)	DD CB DES FE	+
SET 7, (IY+DES)	FD CB DES FE	+
SET 7,A	CB FF	+
SET 7,B	CB F8	+
SET 7,C	CB F9	+
SET 7,D	CB FA	+
SET 7,E	CB FB	+
SET 7,H	CB FC	+
SET 7,L	CB FD	+
SLA (HL)	CB 26	+
SLA (IX+DES)	DD CB DES 26	+
SLA (IY+DES)	FD CB DES 26	+
SLA A	CB 27	+

SLA B	CB 20
SLA C	CB 21
SLA D	CB 22
SLA E	CB 23
SLA H	CB 24
SLA L	CB 25
SRA (HL)	CB 2E
SRA (IX+DES)	DD CB DES 2E
SRA (IY+DES)	FD CB DES 2E
SRA A	CB 2F
SRA B	CB 28
SRA C	CB 29
SRA D	CB 2A
SRA E	CB 2B
SRA H	CB 2C
SRA L	CB 2D
SRL (HL)	CB 3E
SRL (IX+DES)	DD CB DES 3E
SRL (IY+DES)	FD CB DES 3E
SRL A	CB 3F
SRL B	CB 38
SRL C	CB 39
SRL D	CB 3A
SRL E	CB 3B
SRL H	CB 3C
SRL L	CB 3D
SUB (HL)	96
SUB (IX+DES)	DD 96 DES
SUB (IY+DES)	FD 96 DES
SUB A	97
SUB B	90
SUB C	91
SUB D	92
SUB E	93
SUB H	94
SUB L	95
SUB N	D6 XX
XOR (HL)	AE
XOR (IX+DES)	DD AE DES
XOR (IY+DES)	FD AE DES
XOR A	AF
XOR B	A8
XOR C	A9
XOR D	AA
XOR E	AB
XOR H	AC
XOR L	AD
XOR N	EE XX

Y PARA ACABAR EL LIBRO. . ,

Para acabar el libro he hecho un programa realmente interesante. No voy a decir ni lo que hace ni por què lo hace.

Teclèalo <u>tal y como aparece a continuaciòn</u> y pulsa RUN :

```
 10 CLEAR VAL "31769": LET T=VAL "0"
 20 FOR F=VAL "31770" TO VAL "32599": READ A: POKE F,A: LET
        T=T+A: NEXT F
 30 IF T<>VAL "58433" THEN PRINT "ERROR EN DATAS !!!": STOP
 40 INPUT AT VAL "21",VAL "0";AT VAL "5",VAL "5";"INTRODUZCA
        SU NOMBRE";AT VAL "6",VAL "7";"(26 LETRAS MAX.)" ' ' TAB VAL
        " 2 " ; A$
 50 IF LEN A$>VAL "26" THEN GO TO VAL "30"
 60 IF LEN A$=VAL "26" THEN GO TO VAL "90"
 70 FOR F=LEN A$ TO VAL "25"
 80 LET A$=A$+" ": NEXT F
 90 FOR F=VAL "31990" TO VAL "32015": POKE F,CODE A$(F-VAL
        "31990"+VAL "1"): NEXT F
100 RANDOMIZE USR VAL "31770": PAUSE VAL "0"
110 DATA VAL "205" , VAL "107" , VAL "13" , VAL "62" , VAL "2" , VAL
        "205" , VAL "l" , VAL "22"
120 DATA VAL "l" , VAL "91" , VAL "l" , VAL "17" , VAL "69" , VAL "124"
        , VAL "205" , VAL "60" , VAL "32"
130 DATA VAL "17" , VAL "184" , VAL "1" , VAL "33" , VAL "160" , VAL
        "125", VAL "126" , VAL "79" , VAL "35" , VAL "27"
140 DATA VAL "126" , VAL "71" , VAL "35" , VAL "27" , VAL "229" , VAL
        "213" , VAL "205" , VAL "229" , VAL "34" , VAL "209" , VAL "225" ,
        VAL "122" , VAL "179" , VAL "32" , VAL "237" , VAL "201"
150 DATA VAL "22" , VAL "0" , VAL "8" , VAL "16" , VAL "2" , VAL "137"
        , VAL "131" , VAL "22" , VAL "l" , VAL "8" , VAL "139" , VAL "131"
        , VAL "22" , VAL "2" , VAL "8" , VAL "134" , VAL "140" , VAL "16" ,
        VAL "0" , VAL "76"
160 DATA VAL "22" , VAL "2" , VAL "l" , VAL "16" , VAL "2" , VAL "137" ,
        VAL "135" , VAL "22" , VAL "3" , VAL "l" , VAL "142" , VAL "131" ,
        VAL "22" , VAL "4" , VAL "l" , VAL "138" , VAL "142" , VAL "16" ,
        VAL "0" , VAL "69" , VAL "67" , VAL "84" , VAL "79" , VAL "82" ,
        VAL "32" , VAL "68" , VAL "69" , VAL "32" , VAL "76" , VAL "65"
170 DATA VAL "22" , VAL "2" , VAL "14" , VAL "16" , VAL "2" , VAL "137" ,
        VAL "136" , VAL "136" , VAL "22" , VAL "3" , VAL "15" , VAL "138" ,
        VAL "138" , VAL "22" , VAL "4" , VAL "15" , VAL "134" , VAL "134" ,
```

```
            VAL "16" , VAL "0" , VAL "78" , VAL "73" , VAL "86" , VAL "69" ,
            VAL "82" , VAL "83" , VAL "73" , VAL "68" , VAL "65" , VAL "68" ,
            VAL "32" , VAL "68" , VAL "69"
180 DATA VAL "22" , VAL "5" , VAL "3" , VAL "16" , VAL "2" , VAL "137" ,
            VAL "131" , VAL "136" , VAL "22" , VAL "6" , VAL "3" , VAL "138" ,
            VAL "129" , VAL "22" , VAL "7" , VAL "3" , VAL "134" , VAL "140" ,
            VAL "130" , VAL "16" , VAL "0" , VAL "79" , VAL "68" , VAL "73" ,
            VAL "71" , VAL "79"
190 DATA VAL "22" , VAL "5" , VAL "12" , VAL "16" , VAL "2" , VAL "137" ,
            VAL "140" , VAL "132" , VAL "22" , VAL "6" , VAL "13" , VAL "138" ,
            VAL "138" , VAL "138" , VAL "22" , VAL "7" , VAL "13" , VAL "138" ,
            VAL "138" , VAL "142" , VAL "16" , VAL "0" , VAL "65" , VAL "81" ,
            VAL "85" , VAL "73" , VAL "78" , VAL "65"
200 DATA VAL "22" , VAL "9" , VAL "7" , VAL "67" , VAL "69" , VAL "82" ,
            VAL "84" , VAL "73" , VAL "70" , VAL "73" , VAL "67" , VAL "65" ,
            VAL "32" , VAL "81" , VAL "85" , VAL "69"
210 DATA 22 , 9 , 1 , 16 , 2 , 137 , 136 , 22 , 10 , 1 , 138 , 133 , 22 , 11 , 0 , 134 , 134 ,
            137 , 132
220 DATA 22 , 11 , 5 , 16 , 0 , 32 , 32 , 32 , 32 , 32 , 32 , 32 , 32 , 32 , 32 , 32 , 32 , 32 ,
            32 , 32 , 32 , 32 , 32 , 32 , 32 , 32 , 32 , 32 , 32 , 32 , 32
230 DATA 22 , 13 , 3 , 84 , 73 , 69 , 78 , 69 , 32 , 65 , 67 , 82 , 69 , 68 , 73 , 84 , 65 ,
            68 , 79 , 32 , 69 , 78 , 32 , 69 , 83 , 84 , 65 , 22 , 14 , 8 , 85 , 78 , 73 , 86 , 69 , 82 ,
            83 , 73 , 68 , 65 , 68 , 32 , 69 , 76
240 DATA 22 , 16 , 0 , 16 , 2 , 32 , 137 , 139 , 130 , 22 , 17 , 0 , 140 , 32 , 138 , 22 ,
            18 , 2 , 134 , 140
250 DATA 22 , 18 , 4 , 16 , 0 , 73 , 84 , 85 , 76 , 79 , 32 , 68 , 69
260 DATA 22 , 16 , 13 , 16 , 2 , 137 , 131 , 136 , 22 , 17 , 13 , 138 , 129 , 22 , 18 , 13 ,
            134 , 140, 130 , 16 , 0 , 79 , 68 , 73 , 71 , 79
270 DATA 22 , 16 , 21 , 16 , 2 , 137 , 140 , 132 , 22 , 17 , 22 , 138 , 138 , 138 , 22 ,
            18 , 22 , 138 , 138 , 142 , 16 , 0 , 65 , 81 , 85 , 73 , 78 , 65
280 DATA 22 , 20 , 0 , 69 , 108 , 32 , 82 , 101 , 99 , 116 , 111 , 114 , 58
290 DATA 93 , 17 , 94 , 17 , 95 , 17 , 96 , 17 , 97 , 18 , 98 , 18 , 99 , 19 , 100 , 20
300 DATA 100 , 19 , 100 , 18 , 100 , 17 , 100 , 16 , 100 , 15 , 100 , 14 , 100 , 13 , 100 ,
            12 , 100 , 11 , 100 , 10 , 100 , 9
310 DATA 99 , 8 , 99 , 7 , 99 , 6 , 99 , 5 , 99 , 4 , 98 , 4 , 98 , 3 , 97 , 3 , 96 , 3 , 95 , 4 , 94 , 4
320 DATA 94 , 5 , 94 , 6 , 94 , 7 , 94 , 8 , 94 , 9 , 94 , 10
330 DATA 95 , 10 , 95 , 11 , 96 , 11 , 97 , 12 , 98 , 12 , 98 , 13 , 99 , 13
340 DATA 101 , 13 , 102 , 14 , 103 , 14 , 103 , 13 , 104 , 12 , 105 , 12 , 106 , 12 , 107 ,
            12 , 108 , 12 , 108 , 13 , 109 , 12 , 110 , 12
350 DATA 111 , 13 , 111 , 14 , 112 , 15 , 113 , 15 , 114 , 14 , 114 , 13 , 114 , 12 , 113 ,
            12 , 112 , 12 , 115 , 11
360 DATA 116 , 11 , 117 , 12 , 118 , 13 , 119 , 15 , 119 , 14 , 119 , 13 , 120 , 12 , 121 ,
            11 , 122 , 12 , 123 , 12 , 123 , 13 , 123 , 14 , 124 , 13 , 125 , 12 , 126 , 11
370 DATA 134 , 12 , 134 , 13 , 134 , 14 , 134 , 15 , 135 , 16 , 135 , 17 , 135 , 18 , 135 ,
            19 , 135 , 20
380 DATA 136 , 19 , 137 , 18 , 137 , 17 , 138 , 16 , 138 , 15 , 139 , 15 , 140 , 16 , 141 ,
            16 , 142 , 17 , 142 , 15 , 142 , 14 , 142 , 13 , 142 , 12
```

```
390 DATA 145 , 12 , 145 , 11 , 146 , 12 , 146 , 11
400 DATA 154 , 20 , 154 , 19 , 154 , 18 , 155 , 18 , 155 , 17 , 155 , 16 , 156 , 15 , 157 ,
        14 , 157 , 13 , 157 , 12 , 158 , 14 , 158 , 15 , 158 , 16 , 159 , 17 , 159 , 18 , 160 ,
        19 , 161 , 18
410 DATA 162 , 14 , 163 , 14 , 164 , 14 , 165 , 15 , 162 , 15 , 162 , 16 , 163 , 16 , 164 ,
        16 , 161 , 16 , 161 , 15 , 161 , 14 , 161 , 13 , 162 , 12
420 DATA 163 , 12 , 164 , 12 , 165 , 12
430 DATA 168 , 19 , 168 , 18 , 168 , 17 , 168 , 16 , 168 , 15 , 168 , 14 , 168 , 13 , 169 ,
        12 , 170 , 12 , 171 , 12
440 DATA 173 , 13 , 173 , 14 , 174 , 15 , 175 , 15 , 176 , 14 , 176 , 13 , 176 , 12 , 175 ,
        12 , 174 , 12 , 177 , 11
450 DATA 178 , 11 , 179 , 12 , 180 , 13 , 180 , 14 , 181 , 15 , 182 , 14 , 182 , 13 , 183, 12
460 DATA 184 , 12 , 185 , 12 , 188 , 14 , 187 , 14 , 186 , 14 , 185 , 13 , 185 , 12 , 186 ,
        12 , 187 , 12 , 188 , 12 , 189 , 13 , 189 , 14 , 189 , 15 , 189 , 16 , 189 , 17 , 189 , 18
470 DATA 190 , 12 , 191 , 12 , 193 , 14 , 194 , 14 , 195 , 14 , 195 , 15 , 193 , 16 , 194 ,
        16 , 195 , 16 , 192 , 16 , 192 , 15 , 192 , 14 , 192 , 13 , 192 , 12
480 DATA 193 , 12 , 194 , 12 , 195 , 12
490 DATA 130 , 0 , 131 , 0 , 132 , 0 , 133 , 1 , 134 , 1 , 135 , 1 , 135 , 1 , 136 , 1 , 137 ,
        1 , 138 , 2 , 139 , 2 , 140 , 2 , 141 , 2 , 142 , 2 , 143 , 3 , 144 , 3 , 145 , 3 , 146 , 3 ,
        147 , 3 , 148 , 3
```

LIBROS DE INFORMATICA PUBLICADOS POR *PARANINFO S.A.*

Obras Generales
ANGULO.- Introducción a la Informática.
BANKS.- Microordenadores. Cómo funcionan. Para qué sirven.
HUNT .- Manual de informática básica.
SHELLEY .- Microelectrónica.

Diccionarios
HART.- Diccionario del BASIC.
NANIA.- Diccionario de Informática. Inglés. Español. Francés.
OLIVETTI.- Diccionario de informática. Inglés-Español.

Lenguajes
BELLIDO y SANCHEZ. - BASIC para maestros.
BELLIDO y SANCHEZ.- BASIC para estudiantes.
BURKE.- Enseñanza asistida por ordenador .
CHECROUN.- BASIC. Programación de microordenadores.
DELANOY.- Ficheros en BASIC.
GALAN PASCUAL.- Programación con el lenguaje COBOL.
GARCIA MERAYO.- Programación en FORTRAN 77.
LARRECHE.- BASIC. Introducción a la programación.
MARSHALL.- Lenguajes de programación para micros.
MONTEIL.- Primeros pasos en LOGO.
OAKEY.- Lenguaje FORTH para micros.
QUANEAUX.- Tratamiento de textos con BASIC.
ROSSI.- BASIC. Curso acelerado.
SANCHIS LLORCA.- Programación con el lenguaje PASCAL.
VARIOS AUTORES.- 44 Superprogramas en BASIC.
WATT y MANCADA.- BASIC para niños.
WATT y MANCADA.- BASIC avanzado para niños.

Hardware (Equipo Físico)
ANGULO.- Electrónica digital moderna.
ANGULO.- Memorias de Burbujas Magnéticas.
ANGULO.- Microprocesadores. Arquitectura, programación y desarrollo de sistemas.
ANGULO.- Microprocesadores. Curso sobre aplicaciones en sistemas industriales.
ANGULO.- Microprocesadores. Diseño práctico de sistemas.
ANGULO.- Microprocesadores de 16 Bits.
ANGULO.- Prácticas de microelectrónica y microinformática.
ASPINALL.- El microprocesador y sus aplicaciones.
GARLAND.- Diseño de sistemas microprocesadores.
HALSALL y LISTER.- Fundamentos de Microprocesadores.
LEPAPE.- Programación del Z80 con ensamblador.
ROBIN y MAURIN.- Interconexión de microprocesadores.

Spectrum
BELLIDO.- Cómo programar su Spectrum.
BELLIDO.- Cómo usar los colores y los gráficos en el Spectrum.
BELLIDO.- KIT de gráficos para Spectrum.
BELLIDO.- Spectrum Plus Ultra. I.
BELLIDO.- Los trucos del Spectrum.

BELLIDO.- Spectrum. Iniciación al código máquina.
ELLERSHAW y SCHOFIELD.- Las primeras 15 horas con el Spectrum.
ERSKINE.- Los mejores programas para el ZX Spectrum.
WILLIAMS.- Programación paso a paso con el Spectrum.

Sinclair QL
FLEETWOOD.- Sinclair QL.
GALAN PASCUAL.- Programación y práctica con el Sinclair QL.

IBM-PC
MORRILL.- BASIC del IBM.
PLOUIN.- IBM-PC. Características. Programación. Manejo.

Commodore
BISHOP.- Commodore 64. Programas prácticos.
MONTEIL.- Cómo programar su Commodore 64. 2 tomos.
VARIOS AUTORES.- Manejo y programación del Commodore 64.

Apple
ASTIER.- APPLE II y APPLE II/e. Gráficos. Lenguaje. Ensamblador.
ASTIER.- El BASIC del APPLE II y APPLE II/e.

Dragón
BELLIDO.- Amaestra tu DRAGON.
MURRAY.- Programas educativos para el DRAGON 32.

Aplicaciones e Informática Profesional
COHEN.- Estructura. Lógica y diseño de programas.
FLORES.- Estructuración y proceso de datos.
LEWIS y SMITH.- Estructura de datos. Programación yaplicaciones.
LOTT .- Elementos de proceso de datos.
O'NEAL.- Sistemas electrónicos de proceso de datos.
POPKIN y PIKE.- Introducción al proceso de datos.
SCHMIDT y MEYERS.- Introducción a los ordenadores y al proceso de datos.

Técnicas de programación y afines
ABRAMSON.- Teoría de la Información y codificación.
DAX.- CP/M. Guía de utilización.
GAUTHIER y PONTO.- Diseño de programas para sistemas.
HARTMAN, MATTHES y PROEME.- Manual de los sistemas de información. 2 tomos.
LUCAS.- Sistemas de infonnación. Análisis. Diseño. Puesta a punto.

Robótica
ANGULO.- Curso de Robótica.
ANGULO.- Robótica práctica.

VARIOS
ANGULO.- Visión artificial por computador.
ESCUDERO.- Reconocimiento de patrones.
GOSLING.- Códigos para ordenadores y microprocesadores.
PANELL. JACKSON y LUCAS.- El microordenador en la pequeña Empresa.
MARTINEZ VELARDE.- ATARI 520 ST.
PUJOLLE.- Telemática.

EL LIBRO DE CODIGO MAQUINA DEL SPECTRUM

El autor, Juan Martínez Velarde, gran conocedor y articulista sobre el código máquina del ZX Spectrum, cubre con el contenido de esta obra un gran vacío bibliográfico sobre este tema en lengua española.

El texto se divide en cuatro partes bien diferenciadas que permiten al lector escoger su punto de partida en función de sus conocimientos.

Una clara y práctica exposición de los temas, acompañada siempre de programas a modo de ejemplos y ejercicios, permitirán al lector avanzar en el conocimiento del código máquina sin tener que emplear la dureza de otros textos.

Cabe resaltar la oportunidad de programas como el Replicante, Kalidex y otros, que no sólo ayudarán al aprendizaje, sino que servirán en el uso habitual del ZX Spectrum.

El autor, colaborador de ABC y de revistas técnicas especializadas, ha querido, con este atractivo trabajo, animar a los usuarios del ZX Spectrum a adentrarse en el tan apasionante mundo del código máquina.

Si hasta ahora el lector disfrutó con la programación BASIC, a partir de la publicación de esta obra se apasionará con la programación en CODIGO MAQUINA.

Y al final del libro, un programa *sorpresa* para su digitación y posterior estudio y análisis.



Magallanes, 25 - 28015 Madrid

ISBN: 84-283-1349-0